VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

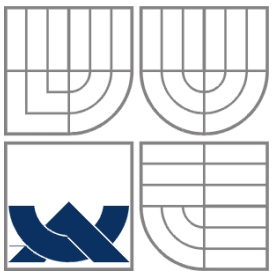
WEBOVÁ APLIKACE ZPROSTŘEDKOVÁVAJÍCÍ VÝSLEDKY TESTOVÁNÍ VÝKONU PLATFORMY JBOSS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

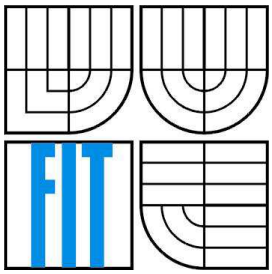
AUTOR PRÁCE
AUTHOR

Bc. JAROSLAV VLASÁK

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

WEBOVÁ APLIKACE ZPROSTŘEDKOVÁVAJÍCÍ VÝSLEDKY TESTOVÁNÍ VÝKONU PLATFORMY JBOSS

TEST RESULT REPOSITORY WITH WEB USER INTERFACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAROSLAV VLASÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2013

Zadání diplomové práce/14625/2012/xvlasa00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2012/2013

Zadání diplomové práce

Řešitel: **Vlasák Jaroslav, Bc.**
Obor: Informační systémy
Téma: **Webová aplikace zprostředkovávající výsledky testování výkonu platformy JBoss**
Test Result Repository with Web User Interface
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s technologiemi pro tvorbu serverových aplikací Java Enterprise Edition (Java EE) a s možnostmi platformě nezávislé komunikace těchto aplikací se vzdálenými aplikacemi.
2. Prostudujte prostředky pro testování výkonu, které se používají při testování platformy JBoss ve firmě Red Hat.
3. Navrhněte aplikaci typu klient-server pro sběr dat z testovacího procesu. Klientská aplikace bude spouštěna spolu s testem a při jeho běhu zasílá výsledky testu serverové aplikaci. Serverová aplikace musí podporovat několik způsobů náhledu na dostupná data prostřednictvím webového klienta a několik operací pro analýzu dat (operace budou určeny podle aktuálních potřeb).
4. Implementujte navrženou aplikaci a otestujte ji ve vhodném testovacím prostředí ve firmě Red Hat.
5. Svou práci zhodnoťte a diskutujte další možná rozšíření.

Literatura:

- E. Jendrock, I. Evans et al.: The Java EE 6 Tutorial: Basic Concepts. ISBN-13: 978-0137081851

Při obhajobě semestrální části diplomového projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Kočí Radek, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 17. září 2012
Datum odevzdání: 22. května 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá vývojem aplikace typu klient-server pro firmu Red Hat. Klientská aplikace se účastní procesu testování platformy JBoss a získává uživatelem definovaná výkonnostní data, která v průběhu testování odesílá serverové aplikaci platformě nezávislou komunikací. Serverová aplikace umožňuje přijatá data analyzovat a několika pohledy vzájemně porovnávat. Tyto služby pro analýzu a srovnávaní dat jsou uživatelům serverové aplikace dostupné prostřednictvím webového klienta. Serverová aplikace rovněž podporuje import výkonnostních dat uložených v XML souboru a také jejich export pro portál qVue. Klientská část aplikace je implementována v jazyce Java a její serverová část je postavena na platformě Java EE.

Abstract

This thesis deals with the development of a client-server application for Red Hat company. Client participates in testing process of JBoss platform and gets user-defined performance data which sends during testing to the server application by platform independent communication. The server application allows to analyze the received data which can be also compared by several perspectives. These services for data analysis and comparison are accessible for server users using the web client. The server application supports the import of the performance data stored in the XML file and also their export for qVue portal. The client part of the application is implemented in Java and the server application is based on Java EE platform.

Klíčová slova

testování výkonu, jednotkové testování, JUnit, TestNG, Maven, Maven Surefire plugin, Java EE, JPA, EJB, Facelets, JSF, HTTP, RESTful webová služba, JAX-RS, PrimeFaces, JBoss Application Server, PostgreSQL, OpenShift

Keywords

performance testing, unit testing, JUnit, TestNG, Maven, Maven Surefire plugin, Java EE, JPA, EJB, Facelets, JSF, HTTP, RESTful web service, JAX-RS, PrimeFaces, JBoss Application Server, PostgreSQL, OpenShift

Citace

Vlasák Jaroslav: Webová aplikace zprostředkovávající výsledky testování výkonu platformy JBoss, diplomová práce, Brno, FIT VUT v Brně, 2013

Webová aplikace zprostředkovávající výsledky testování výkonu platformy JBoss

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Kočího, Ph.D.
Další informace mi poskytl Mgr. Libor Zoubek, zástupce firmy Red Hat.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jaroslav Vlasák
21. května 2013

Poděkování

Tímto bych rád poděkoval Ing. Radku Kočímu, Ph.D. a Mgr. Liboru Zoubkovi za odborné vedení mé práce, cenné připomínky a čas, který mi věnovali.

Obsah

Obsah 1
1 Úvod 4
2 Automatizované testování výkonu 6
2.1 Jednotkové testování 6
2.1.1 JUnit 7
2.1.2 TestNG 8
2.2 Testování výkonu 8
2.2.1 Výkonnostní testy 9
2.2.2 Výkonnostní testy podstatné z hlediska návrhu 10
2.2.3 Výkonnostní testy platformy JBoss 11
3 Technologie použité pro vývoj 12
3.1 Technologie klientské aplikace 12
3.1.1 Knihovna SIGAR 12
3.1.2 Maven 12
3.1.3 Maven Surefire Plugin 15
3.1.4 Gson 16
3.1.5 Response-Time Filter 17
3.1.6 SLF4J a LOG4J 17
3.2 Technologie serverové aplikace 18
3.2.1 Java Persistence API 18
3.2.2 Enterprise Java Beans 19
3.2.3 Servlet 20
3.2.4 Java Server Faces 20
3.2.5 Facelets 21
3.2.6 PrimeFaces 22
3.2.7 Architektura typu REST 22
3.2.8 Aplikační server 24
3.2.9 JBoss AS7 Deployment Plugin 25
3.2.10 PostgreSQL 25
3.2.11 JBoss Tools 26
3.2.12 OpenShift 26
4 Realizace klientské části aplikace 27
4.1 Analýza a návrh klientské aplikace 27
4.1.1 Případy užití klienta 27

4.1.2 Módy klienta 28
4.1.3 Paměť klienta 29
4.1.4 API klienta 29
4.1.5 Listenery klienta 29
4.1.6 Komunikace klienta s ostatními entitami 31
4.1.7 Inicializace vzdáleného klienta a princip řešení jeho komunikace 31
4.1.8 Návrh struktury klienta 32
4.2 Analýza a návrh komponent klientské aplikace 34
4.3 Implementace klientské části aplikace 34
4.3.1 Realizace klienta 35
4.3.2 Realizace komponent klienta 35
4.3.3 Výsledná struktura klientské části aplikace 37
5 Realizace serverové aplikace 38
5.1 Analýza a návrh 38
5.1.1 Srovnávací pohledy na výkonnostní data 38
5.1.2 Diagram případů užití 39
5.2 Implementace 41
5.2.1 Vrstvy serverové aplikace 41
5.2.2 Realizace perzistentní vrstvy 42
5.2.3 Realizace business vrstvy a REST architektury 44
5.2.4 Realizace prezentační vrstvy 47
6 Komunikační protokol 51
6.1 Analýza a návrh 51
6.1.1 Volba technologií pro komunikaci 51
6.1.2 Princip řešení komunikace 51
6.2 Implementace 52
6.2.1 Komunikace mezi klientem a serverovou aplikací 52
6.2.2 Komunikace portálu qVue se serverovou aplikací 55
7 Testování projektu 57
7.1 Příprava testovacího prostředí 57
7.2 První fáze testování 59
7.3 Druhá fáze testování 59
7.3.1 Popis a princip testování 60
7.3.2 Výsledky distribuovaného testování 61
7.4 Třetí fáze testování 64
8 Závěr 66
Seznam příloh 71

Příloha A - Specifikace požadavků....72
Příloha B – Diagram tříd klientské aplikace....75
Příloha C – Databázové schéma serverové aplikace....76
Příloha D – Ukázka náhledu na stránku view1.xhtml (pohled typu I)....77
Příloha E – Ukázka náhledu na stránku view2.xhtml (pohled typu II)....78
Příloha F – Ukázka náhledu na stránku view3.xhtml (pohled typu III)....79
Příloha G – Manuál pro klientskou aplikaci....80
Příloha H – Manuál pro serverovou aplikaci....86
Příloha I – Obsah přiloženého DVD....90

1 Úvod

Tato práce, jejímž zadavatelem je firma Red Hat, se zabývá realizací klient-server aplikace pro sběr a srovnávání výkonnostních dat z testovaných verzí projektů (tzv. buildů) platformy JBoss. Potřeba vzniku pro tuto aplikaci vyplynula ze současné situace ve firmě, kdy aktuálně používaná aplikace pro tyto účely již nedostačuje. Hlavními nedostatky v současné době používané aplikace jsou zejména prezentace výsledků výkonnostního testování pouze statickým způsobem a nemožnost výkonnostní data předávat portálu qVue (slouží pro řízení kvality vyvíjených projektů), který využívá firemní Quality Assurance (QA) oddělení. Vyvíjená aplikace si klade za cíl tyto nedostatky odstranit a mimo to bude postavena na novějších technologiích.

Klientská aplikace (klient) bude moci v průběhu jednotkových JUnit či TestNG testů platformy JBoss získávat uživatelem definovaná výkonnostní data a odesílat je platformě nezávislou komunikací serverové aplikaci. Jednotkové testy budou spouštěny v rámci tzv. test suit, které budou testovat různé buildy projektů na různých testovacích platformách (testovací platforma je určena hardwarem počítače a jeho operačním systémem), přičemž jednotlivých testů se bude moci účastnit i více klientů, kdy každý bude získávat a odesílat svá specifická výkonnostní data.

Serverová aplikace bude přijatá data trvale uchovávat a prostřednictvím webového klienta je umožní analyzovat pomocí několika pohledů, které budou výkonnostní data prezentovat ve formě grafů a tabulek. Pohledy umožní snadné srovnání výkonu mezi test suitami či jejich metodami u zvolených buildů z hlediska stanovených výkonnostních atributů a to i s ohledem na platformy, na nichž testování probíhalo. Serverová aplikace tak bude své uživatele informovat o výkonnostním vývoji jednotlivých buildů projektů, čímž bude poskytovat cennou zpětnou vazbu pro jejich další vývoj, a umožní tak vytvářet výkonnostně odladěný software.

Práce je rozčleněna do osmi kapitol. Na úvod navazuje kapitola 2, která vysvětluje pojem automatického testování výkonu a s tím související jednotkové testování. Dále jsou představeny některé typy výkonnostních testů a je popsán princip a ukázka výkonnostního testování platformy JBoss. Kapitola 3 stručně seznamuje s technologiemi, které budou použity pro vývoj klientské i serverové aplikace. Kapitola 4 se zabývá analýzou, návrhem a implementací klientské aplikace. V kapitole 5 je analogicky jako u klientské aplikace popsán vývoj serverové aplikace. Kapitola 6 rozebírá možnosti platformě nezávislé komunikace mezi klientem a serverovou aplikací a popisuje návrh a realizaci této komunikace s využitím zvolené technologie. Kapitola 7 popisuje testování projektu. Nejprve je zde popsána příprava prostředí pro testování a poté všechny tři fáze testování. Závěrečná kapitola hodnotí použitelnost vyvinuté klient-server aplikace v praxi a také navrhuje další rozšíření, jež by bylo vhodné do její serverové části dále zapracovat. V přílohách práce jsou shrnuty požadavky kladené zadavatelem na vyvíjenou klient-server aplikaci. Dále je to implementační diagram tříd klientské aplikace, databázové schéma používané serverovou aplikací, ukázky

realizovaných pohledů serverové aplikace pro výkonnostní srovnávání a stručný uživatelský manuál k oběma aplikacím.

2 Automatizované testování výkonu

Pod pojmem automatizované testování výkonu si lze představit jednotkové testy, v nichž jsou získávána výkonnostní data. Takto upravené jednotkové testy mají výrazně přidanou hodnotu a jsou proto neocenitelným pomocníkem při vývoji kvalitního a výkonnostně optimalizovaného software.

Vlastní získávání výkonnostních dat má být realizováno klientskou aplikací napsanou v jazyce Java, jejíž implementace je jedním z cílů této práce. Tato aplikace má být schopna pracovat v rámci jednotkových testů napsaných s použitím knihoven JUnit a TestNG. Zde zmíněné pojmy popíši blíže v následujících kapitolách.

2.1 Jednotkové testování

Jednotkové testování je doslovný překlad slovního spojení unit testing, které se na rozdíl od zmíněného českého překladu v programování používá běžně. Další možné překlady tohoto spojení jsou testování jednotek či testování aplikačních jednotek [1].

Jednotkou se rozumí nejmenší fragment zdrojového kódu, který může být přeložen a spuštěn. Testováním těchto jednotek se zjišťuje, zda fungují podle našich požadavků. Ověření jejich funkčnosti se provádí tak, že se výsledek získaný testováním porovná s předpokládanou hodnotou. Testování každé jednotky se provádí izolovaně, tak aby byly všechny testy na sobě nezávislé a vzájemně se neovlivňovaly. Z hlediska procedurálního programování se jednotkou chápe program, funkce či procedura. Z pohledu objektově-orientovaného je jednotkou třída, u níž se samostatně testují její metody, resp. členské funkce.

Jednotkové testy hrají klíčovou roli při vývoji řízeným testy (ang. test driven development). Tato metodika vyžaduje nejprve napsání jednotkových testů a až poté se začne s implementací. Testy se pak automaticky vykonávají v iteracích vždy po přidání nové funkcionality u vyvíjeného software. Snadno a rychle se tak ověří funkcionalita, což ušetří mnoho času při hledání chyb, které se nově podařilo zanést do kódu.

Tento typ testů vyžaduje hlouběji porozumět zdrojovému kódu, proto je zpravidla píší programátoři, než testeři [2].

Výhody jednotkového testování [3]:

- je automatizované,
- rychle ověří funkcionalitu při změně implementace či refaktorizaci,
- zjednodušuje integrační testy (návrh zdola-nahoru),
- z testů lze vyčíst funkcionalitu (funkce dokumentace).

Mezi nevýhody patří:

- větší pracnost během vývoje,
- testuje pouze funkční chyby (úspěšnost závisí především na pokrytí všech možností),
- neodhalí chyby integrační, výkonnostní a systémové,
- může pouze ukázat přítomnost chyby, nezaručí však, že je software bez chyb.

2.1.1 JUnit

JUnit je open-source rámec (framework) pro jednotkové testování napsaný v jazyce Java. Jeho autoři jsou Kent Beck a Erich Gamma [4].

JUnit slouží výhradně pro testování metod javovských tříd. Jeho implementace vychází z frameworku SUnit, který byl určen pro programovací jazyk SmallTalk. Z rámce SUnit vycházejí i ostatní frameworky pro jednotkové testování, které se souhrnně označují xUnit. Díky tomuto faktu je SUnit považován za otce všech těchto frameworků [5]. V tabulce 1 jsou uvedeni někteří další zástupci rodiny xUnit spolu s programovacími jazyky, pro něž jsou určeny.

Programovací jazyk	**Unit framework**
C	CUnit
C++	CPPUnit
C#	NUnit
Perl	Test::Class, Test::Unit
Python	PyUnit
PHP	PHPUnit
Haskell	HUnit

Tabulka 1: Zástupci rodiny xUnit

Verze JUnit

V současné době lze vývojové verze frameworku JUnit rozdělit do 2 skupin. Jsou to všechny verze JUnit až do verze 3.8 (dále jen JUnit verze 3.8) a JUnit verze 4.0 a vyšší (dále jen JUnit verze 4.0). Tyto verze se vzájemně liší svou implementací a také použitím [4].

JUnit verze 3.8 vyžaduje, aby jeho testované třídy byly potomky třídy *junit.framework.TestCase*. Názvy testovaných metod pak musí mít prefix *test*, jinak budou při testovaní vynechány.

JUnit verze 4.0 využívá kompletně anotací. Testované metody nemusí začínat s prefixem *test*, ale musí být anotovány anotací *@Test*. Testované třídy již nejsou potomky třídy

junit.framework.TestCase. Oproti verzi 3.8 umožňuje tato verze JUnit specifikovat vlastní JUnit listener [6].

JUnit listener

JUnit verze 4.0 standardně při testování používá listener, který je reprezentován třídou *RunListener* z jejího balíčku *org.junit.runner.notification*. Kromě tohoto listeneru umožňuje JUnit použít také vlastní listener. Ten lze vytvořit tak, že vytvoříme třídu, která bude dědit z třídy *RunListener* a přepíšeme její metody dle vlastní potřeby. Detailní informace ke třídě *RunListener* a jejím metodám lze nalézt v programové dokumentaci této třídy [7].

2.1.2 TestNG

TestNG je testovací framework nové generace, který vychází z JUnit a navíc jeho funkcionalitu rozšiřuje o nové rysy (seskupování testů do skupin, parametrizované testování s využitím objektů, závilostní testování apod.). Na rozdíl od JUnit TestNG podporuje všechny typy testů. Je také jednodušší z hlediska použití a porozumění [8]. Dá se tedy přepokládat, že někdy v budoucnu JUnit zcela nahradí.

TestNG listener

Stejně jako JUnit i TestNG umožňuje definovat vlastní listener pro testování. Jako listener lze použít každou třídu, která implementuje rozhraní *ITestNGListener* z balíčku *org.testng*. Kromě tohoto rozhraní lze použít pro implementaci vlastního listeneru ještě další rozhraní, která rozhraní *ITestNGListener* rozšiřují (*ITestListener*, *IConfigurable*, *IHookable*, *IReporter*, *ISuiteListener*, a další) [8]. Z těchto zmíněných rozhraní je pro testování přímo určeno rozhraní *ITestListener*. Jeho programovou dokumentaci lze nalézt zde [9].

2.2 Testování výkonu

Testování výkonu je nezbytným nástrojem pro odhalení výkonnostně slabých míst softwaru (tzv. bottlenecks), které jsou hlavní příčinou celkového zpomalení systému. Optimalizací těchto slabých míst se pak dosáhne výrazně lepšího výkonu u vyvíjených aplikací.

Výkon je velmi důležitým hlediskem k posuzování kvality software, proto je také nedílnou součástí metodiky FURPS[1]. Mohli bychom namítnout, že sledování výkonu není důležité. Výkon počítačů přece neustále roste. Rostou však i naše požadavky na výpočetní výkon, kdy počítáme stále složitější úlohy, simulujeme komplikované jevy a běžně používáme několik aplikací současně.

[1] http://cs.wikipedia.org/wiki/FURPS

Webové aplikace se pak potýkají s neustále narůstajícím počtem souběžně přistupujících uživatelů, kdy každá nepatrná prodleva při vyřízení požadavku při tak obrovském množství uživatelů znamená velké zpoždění a aplikace se stane nepoužitelnou.

Je proto žádoucí během vývoje každého projektu tyto projekty výkonnostně testovat, ale také srovnávat jejich jednotlivé buildy a případné výkonnostní odchylky co nejdříve identifikovat a optimalizovat. Tímto způsobem budeme vytvářet kvalitnější software, výrazně zefektivníme vývoj a snížíme vynaložené finanční prostředky.

2.2.1 Výkonnostní testy

Výkonnostních testů existuje několik typů. Zde jsou uvedeny některé z nich [10]:

Load test

Nejjednodušší forma výkonnostního testování. Tento test se obvykle provádí, aby se zjistilo, jak se aplikace bude chovat pod určitým zatížením. U webových aplikací je toto zatížení typicky simulováno souběžným přístupem virtuálních uživatelů na server, kteří provádí hromadně zvolené operace.

Stress test

Cílem tohoto testu je nalézt maximální kapacitu u testované oblasti systému, při které je systém ještě schopen pracovat. Test pracuje tím způsobem, že neustále zvyšuje u testované oblasti systému zátěž, až dojde k jeho pádu. Příkladem mohou být neustále rostoucí výpočetní nároky na operační paměť.

Soak test

Jedná se o test, který zjišťuje, zda u aplikace nenastanou problémy při jejím dlouhodobém očekávaném zatížení. Většinou se během tohoto testu monitoruje využitá paměť. Ta odhalí případné paměťové úniky, které mohou vznikat při opakovaném volání testované operace. Test se zaměřuje také na odezvu systému, jež by se postupem času neměla zhoršovat.

Spike test

Vyvíjí na systém zátěž, která se rapidně mění. Cílem je zjistit, jestli se systém s těmito zátěžovými výkyvy vypořádá a jakým způsobem.

Configuration test

Testuje, jaké dopady má na výkon systému změna jeho konfigurace.

Isolation test

Používá se pro izolaci a potvrzení chyby v systému pomocí opakovaného provedení testu, který chybu způsobuje.

2.2.2 Výkonnostní testy podstatné z hlediska návrhu

Pro návrh řešení klientské ani serverové aplikace není podstatné, jaké typy výkonnostních testů se s jejich využitím budou provádět. Je však nezbytné uvažovat a předvídat různě komplikované způsoby testování, které mohou být s jejich využitím v praxi realizovány.

Z požadavků, které zadavatel klade na vyvíjenou klient-server aplikaci, a uvedených typů výkonnostních testů lze charakterizovat tři typy testů, které jsou vzájemně odlišné a důležité z hlediska návrhu.

Jednorázový test

Tyto typy testů jsou časově nenáročné. Jejich výsledkem je často pouze jediná hodnota u měřených výkonnostních atributů, a proto lze naměřené hodnoty zasílat až při dokončení testu. Příkladem takového testu může být např. celkový procesorový čas spotřebovaný při nějaké databázové operaci nebo určení doby odezvy webového serveru (tzv. response time) apod.

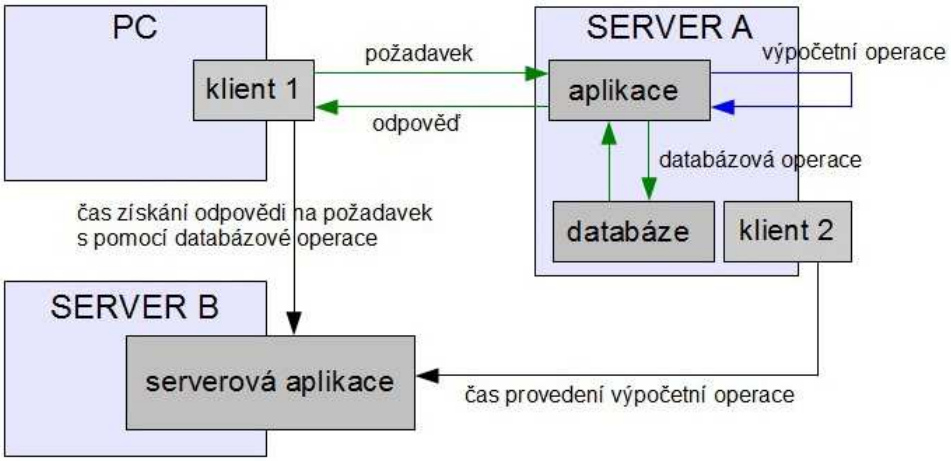
Dlouhodobý test

Je časově podstatně náročnější než jednorázový test a tudíž je vhodné naměřené výsledky odesílat průběžně. Výstupem testu je sada hodnot, u které nás zajímá především její časový průběh. Příkladem tohoto typu testu může být postupné posílání požadavků na server v rámci jednoho testu za účelem zjištění všech odezev serveru.

Distribuovaný test

Jedná se o nejsložitější typ testu, kdy se jednoho testovacího případu účastní více klientů na různých fyzických počítačích. Přepokládejme, že cílem testu je určení času stažení nějaké webové stránky (od vyslání požadavku na tuto stránku až po její obdržení) a dále určení času potřebného pro vykonání jisté výpočetní operace. Tato operace se přitom provádí nad daty získanými z databáze před vlastním generováním stránky.

Situaci popisuje obrázek 1, kde klient 1 (umístěn na počítači PC) zjišťuje čas potřebný pro stažení webové stránky aplikace, která je nasazena na počítači SERVER A, a klient 2 (umístěn na počítači SERVER A) určuje čas výpočetní operace. Po dokončení měření oba klienti odesílají naměřené hodnoty serverové aplikaci nasazené na počítači SERVER B.



Obrázek 1: Ukázka distribuovaného testování

Díky tomuto přístupu testování je možné identifikovat tzv. bottlenecky i u složitějších aplikací (např. několika propojených webových služeb).

2.2.3 Výkonnostní testy platformy JBoss

Výkonnostní testy platformy JBoss většinou přesně nepokrývají výše uvedené typy výkonnostních testů, ale často vzniknou jejich nejrůznějším zkřížením. Z tohoto důvodu bude vhodné uvést nějaký komplexnější příklad testování této platformy, jež by měla vyvinutá aplikace plně pokrýt.

Příklad výkonnostního testu platformy JBoss

Předpokládejme, že máme jednotkový test a v něm vytváříme N požadavků na určitou webovou stránku aplikace, která je nasazena na vzdáleném serveru. U jednotkového testu budeme zjišťovat celkový spotřebovaný procesorový čas, změnu JVM[2] paměti a dále aktuální hodnotu využité JVM paměti po každém vykonaném požadavku.

Na vzdáleném serveru pak bude pro každý příchozí požadavek měřen čas odezvy serveru a dále reálný čas, který je potřebný pro vykonání databázové operace, jež pro požadovanou stránku získává data. Po vygenerování požadované stránky bude na serveru navíc ještě určena aktuální hodnota využité JVM paměti.

[2] Java Virtual Machine, http://cs.wikipedia.org/wiki/Java_Virtual_Machine

3 Technologie použité pro vývoj

Tato kapitola stručně popisuje technologie, které budou použity při vývoji klientské i serverové aplikace.

3.1 Technologie klientské aplikace

3.1.1 Knihovna SIGAR

Tato knihovna poskytuje jednotné API[3] pro přístup k systémovým informacím v různých operačních systémech. SIGAR v současné době obsahuje podporu pro operační systém Linux, FreeBSD, Windows, Solaris, AIX, HP-UX a Mac OSX a to bez ohledu na verzi tohoto systému a hardwarovou platformu, na které je systém provozován. Aby bylo možno knihovnu v daných operačních systémech používat, je potřeba programu s jejím voláním předat cestu k jejím nativním knihovnám. V jazyce Java to lze učinit nastavením systémové vlastnosti *java.library.path*.

Samotné jádro knihovny je naprogramováno v jazyce C a kromě Javy ji lze volat také z jazyka Perl a C#. Knihovna je pod licencí Apache 2.0 [11].

3.1.2 Maven

Nástroj Maven slouží pro automatizované sestavování aplikací, pro které využívá uniformní buildovací systém. Používá se především u projektů v jazyce Java, ale podporuje i jiné jazyky. Mezi další klady Mavenu patří snadná správa externích knihoven projektu (tzv. závilostí). Maven dále poskytuje celou řadu funkcionalit, které souvisí s vývojem aplikací. Většina z nich je dostupná prostřednictvím pluginů, které lze do projektu založeném na Mavenu přidat.

Maven projekt musí ve svém kořenovém adresáři obsahovat kromě standardních adresářů také soubor *pom.xml* (tzv. POM). Zkratka POM znamená projektový objektový model a jedná se o xml reprezentaci projektu, kde se nacházejí všechny jeho metadata [12]. Kompletní struktura *pom.xml* souboru je zachycena v příkladu 1.

[3] Application Programming Interface, http://cs.wikipedia.org/wiki/API

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                           http://maven.apache.org/xsd/maven-4.0.0.xsd">
       <modelVersion>4.0.0</modelVersion>

       <!-- Základní informace o projektu -->
       <groupId>cz.vutbr.fit.mis.dip</groupId>
       <artifactId>PerfClient</artifactId>
       <version>1.0</version>
       <packaging>jar</packaging>
       <dependencies>...</dependencies>
       <parent>...</parent>
       <dependencyManagement>...</dependencyManagement>
       <modules>...</modules>
       <properties>...</properties>

       <!-- Nastavení pro buildování -->
       <build>...</build>
       <reporting>...</reporting>

       <!-- Více informací o projektu -->
       <name>...</name>
       <description>...</description>
       <url>...</url>
       <inceptionYear>...</inceptionYear>
       <licenses>...</licenses>
       <organization>...</organization>
       <developers>...</developers>
       <contributors>...</contributors>

       <!-- Nastavení prostředí -->
       <issueManagement>...</issueManagement>
       <ciManagement>...</ciManagement>
       <mailingLists>...</mailingLists>
       <scm>...</scm>
       <prerequisites>...</prerequisites>
       <repositories>...</repositories>
       <pluginRepositories>...</pluginRepositories>
       <distributionManagement>...</distributionManagement>
       <profiles>...</profiles>
</project>
```

Příklad 1: Kompletní struktura pom.xml. souboru [13]

Nepostradatelnými značkami v dokumentu jsou značky s názvem *groupId*, *artifactId* a *version* (ostatní značky jsou nepovinné). *GroupId* je obecně jednoznačný název organizace či projektu, v rámci nějž je aplikace vyvíjena. *ArtifactId* značí název projektu a *version* jeho verzi. Hodnoty těchto značek určují umístění a název zkompilovaného projektu v lokálním repozitáři Mavenu (dle uvedeného příkladu, tedy: *cz/vutbr/fit/mis/dip/PerfClient-1.0.jar*). Pokud projekt

vyžaduje externí knihovny, jsou specifikovány uvnitř sekce <*dependencies*>. Pro knihovnu TestNG ve verzi 6.5.1 bude závilost definována takto:

```
<dependency>
     <groupId>org.testng</groupId>
     <artifactId>testng</artifactId>
     <version>6.5.1</version>
</dependency>
```

Externí knihovny (závilosti) Maven automaticky vyhledává v jeho repozitářích. Jakmile danou knihovnu nalezne, stáhne ji do lokálního repozitáře a následně ji vloží do kompilovaného projektu.

Maven rozlišuje tyto repozitáře:

- **Lokální** – V linuxu je umístěn v adresáři *~/.m2* lokálního počítače a obsahuje všechny stažené externí knihovny a projekty, které jsme do něj instalovali.
- **Vzdálený** – Může sloužit jako firemní uložiště všech používaných knihoven. Cesta k němu se definuje v sekci <*repositories*> souboru *pom.xml*.
- **Centrální** – Obsahuje všechny externě publikované knihovny[4].

Vyhledávání externí knihovny probíhá tak, že ji Maven nejprve hledá v lokálním repozitáři. Pokud ji zde nenalezne, zkusí ji najít v centrálním repozitáři. Pokud ji nenalezne ani tady, začne ji hledat ve vzdáleném repozitáři (pokud je definován). V případě, že Maven knihovnu nenalezne, nastane chyba.

Maven a jeho životní cykly

Maven obsahuje tři vestavěné životní cykly. Jsou to cykly *clean*, *site* a *default*. Cyklus *clean* slouží pro smazání všech souborů, které vznikly při kompilaci. Cyklus *site* pak zahrnuje vygenerování projektové dokumentace a její případné umístění na webový server. Cyklus *default* je nejdůležitější, a proto popsán více.

Default cyklus

Cyklus *default* je srdcem každého Maven projektu, protože pokrývá proces kompilování, testování a distribuci artefaktu (zkompilovaného projektu) včetně jeho nasazení na aplikační server. Obsahuje více jak 20 fází, z nichž nejdůležitější jsou uvedeny v tabulce 2.

[4] dostupné na: http://search.maven.org

Název fáze	Popis fáze
validate	Validuje dostupnost a korektnost všech projektových informací.
compile	Kompiluje zdrojové kódy projektu.
test	Spustí jednotkové testy s využitím definovaného frameworku.
package	Ze zkompilovaného kódu vytvoří tzv. build (jar nebo war soubor).
integration-test	Otestuje build v integračním testovacím prostředí.
verify	Spustí verifikaci nad buildem, zda je validní.
install	Instaluje build do lokálního repozitáře.
deploy	Instaluje finální build do vzdáleného repozitáře.

Tabulka 2: Nejdůležitější fáze default cyklu Mavenu

Ke spuštění určité fáze Mavenu stačí zadat příkaz Mavenu (tedy *mvn*) a dále uvést název požadované fáze. Maven pak vykoná všechny fáze, které zadanou fázi předcházely včetně uvedené. Pokud tedy budeme chtít např. vytvořit build Maven projektu, stačí v jeho kořenovém adresáři zadat příkaz: *mvn package*. Maven pak vykoná v daném pořadí fázi *validate*, *compile*, *test* a *package* a jeho výsledkem bude build projektu.

Případné Maven pluginy se vkládají mezi značky <*plugins*> v sekci <*build*> souboru *pom.xml*. Definici každého pluginu pak musí být mezi značkami <*plugin*>.

3.1.3 Maven Surefire Plugin

Surefire plugin není použit pro vývoj klientské aplikace, ale je nutné jej integrovat do projektu, který je pomocí klienta testován. Tento plugin totiž umožňuje definovat vlastní listener pro JUnit (od verze 4.0) a TestNG testy, což je pro samotné fungování klienta nezbytný krok.

Surefire plugin se aktivuje během test fáze životního cyklu Mavenu, kdy spouští jednotkové testy. Za testy požaduje všechny soubory v adresáři s java testy a všech jeho podadresářích, pro jejichž název platí, že začíná nebo končí řetězcem “Test” nebo končí řetězcem “TestCase”. Z testování generuje také reporty, které jsou umístěny do adresáře *target/surefire-reports* v kořenovém adresáři projektu [14].

Vlastní listener lze pro testovaný projekt definovat dvěma způsoby:

1) Použitím Suite XML souboru

Tento způsob je podporován jen knihovnou TestNG. Vlastní listener je specifikován v Suite XML souboru (v příkladu 2 pojmenován jako *testng.xml*), na který se v sekci <*configuration*> Maven Surefire pluginu odkážeme dle příkladu 2.

```
<configuration>
	<suiteXmlFiles>
		<suiteXmlFile>testng.xml</suiteXmlFile>
	</suiteXmlFiles>
</configuration>
```

Příklad 2: Ukázka definování vlastního listeneru prostřednictvím Suite XML souboru

2) Přímým definováním listeneru

Listener se v tomto případě definuje v sekci <*configuration*> Maven Surefire pluginu dle ukázky v příkladu 3. V sekci <*property*> se uvnitř značky <*name*> definuje jméno listeneru a uvnitř značky <*value*> samotná třída listeneru.

```
<configuration>
	<properties>
		<property>
			<name>listener</name>
			<value>com.company.MyListener</value>
		</property>
	</properties>
</configuration>
```

Příklad 3: Ukázka specifikace listeneru přímým definováním

U projektu, který má jedním z výše uvedených způsobů definován vlastní listener, poté stačí zadat příkaz *mvn test* a testování projektu s vlastním listenerem je započato.

3.1.4 Gson

Gson je Java knihovna, která umožňuje serializovat Java objekty do jejich JSON[5] textové reprezentace a naopak (tedy deserializovat). Serializované objekty mohou obsahovat další objekty, které mohou být různě složité. Gson umožňuje serializovat také např. vnitřní třídy, kolekce a null objekty. Serializované objekty nemusí obsahovat žádné anotace a není tedy nutný žádný zásah do kódu. Touto vlastností se knihovna Gson odlišuje od většiny ostatních podobných knihoven. Pro serializaci a deserializaci se používají metody s názvem *toJson* a *fromJson* třídy Gson [15]. Ukázka serializace objektu do textové reprezentace ve formátu JSON a jeho zpětná deserializace do původního objektu je s použitím této knihovny uvedena v příkladu 4.

[5] JavaScript Object Notation, http://www.json.org

```
AttrResultData data = new AttrResultData();
// zde bude objekt data naplněn daty
// vytvoření instance gson
Gson gson = new Gson();
// serializace
String jsonData = gson.toJson(data);
// deserializace
AttrResultData result = gson.fromJson(jsonData, AttrResultData.class);
```

Příklad 4: Ukázka serializace a zpětné deserializace objektu data třídy AttrResultData

3.1.5 Response-Time Filter

Response-Time Filter[6] je servletová aplikace , která umožňuje měřit dobu odezvy (tzv. response time) webového serveru u Java EE aplikací. Tato aplikace je součástí projektu se zkratkou RHQ, který spadá do JBoss technologií. Aplikace dobu odezvy měří v metodě s názvem *doFilter* servletu *RtFilter* a zjištěné hodnoty loguje do souboru. Pro vlastní logování pak poskytuje několik možností nastavení (viz. [16]).

Použití servletu RtFilter

Aby mohl tento servlet provádět svou funkci, je potřeba knihovnu celé jeho aplikace přidat do projektu Java EE aplikace a filtr definovat v jejím souboru *web.xml*. Nejjednodušší způsob definice filtru je uveden na příkladu 5. V tomto případě měří filtr response time pro všechny webové stránky aplikace (určeno obsahem sekce <*url-pattern*>). Jméno filtru je definováno v sekci <*filter-name*> a samotná třída filteru v sekci <*filter-class*>.

```
<filter>
	<filter-name>RHQRtFilter</filter-name>
	<filter-class>org.rhq.helpers.rtfilter.filter.RtFilter</filter-class>
</filter>
<filter-mapping>
	<filter-name>RHQRtFilter</filter-name>
	<url-pattern>/*</url-pattern>
</filter-mapping>
```

Příklad 5: Ukázka definice servletu RtFilter pro měření response time v souboru web.xml

3.1.6 SLF4J a LOG4J

SLF4J

SLF4J je jednoduchá fasáda, resp. abstrakce, pro nejrůznější logovací knihovny jako je *java.util.logging*, *log4j* a *logback*. SLF4J využívá rozhraní *Logger*, které definuje obecné metody pro

[6] zdrojové kódy verze 4.3.0 jsou dostupné na:
http://grepcode.com/snapshot/repository.jboss.org/nexus/content/repositories/releases/org.rhq/rhq-rtfilter/4.3.0/

výpis logů. Tyto metody volají logovací metody konkrétní použité logovací knihovny [17]. Ukázka použití SLF4J je na příkladu 6, kdy získáme logger s názvem *HelloWorld* a pomocí metody s názvem *info* vypíšeme při užití stejnojmenné hladiny logování zadanou hlášku.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld{
	public static void main(String[] args) {
		Logger logger = LoggerFactory.getLogger(HelloWorld.class);
		logger.info(“Hello world”);
	}
}
```

Příklad 6: Ukázka použití SLF4J pro logování

LOG4J

LOG4J je populární logovací knihovna určena pro jazyk Java. Oproti jiným logovacím knihovnám umožňuje např. povolit, resp. měnit konfiguraci, logování za běhu aplikace a to bez nutnosti modifikace binárního kódu vlastní aplikace. Této funkcionality je dosaženo prostřednictvím konfigurační souboru, kde lze nastavit veškeré detaily logování (hladinu logování, kam se bude logovat, formát logů apod.) [18].

3.2 Technologie serverové aplikace

Serverová aplikace bude vyvinuta na platformě Java EE a pro komunikaci s klienty bude disponovat architekturou typu REST. Z platformy Java EE, která je velmi rozsáhlá, jsou zde stručně představeny pouze technologie Java Persistence API, Enterprise Java Beans, Java Server Faces a Facelets. JAX-RS, které je také součástí Javy EE, je popsáno v rámci REST[7] architektury. Kromě těchto technologií jsou zde popsány ještě některé další. Pro seznámení se s platformou Java EE byl použit zdroj [19].

3.2.1 Java Persistence API

Java Persistence API neboli JPA je specifikační standard pro objektově relační mapování, který popisuje:

- JPA
- dotazovací jazyk
- Java Persistence Criteria API
- metadata pro objektově relační mapování

[7] Representational State Transfer

Objekty, které lze ukládat do databáze, se nazývají entity. Entity reprezentují tabulku v relační databázi, kdy každá její instance odpovídá jednomu řádku v tabulce a každý její atribut jednomu sloupci v tomto řádku. Aby mohla být entitní třída uložena do relační databáze tímto způsobem, musí mít bezparametrický konstruktor, anotaci *javax.persistence.Entity* a její atributy musí disponovat getterem a setterem. Vztahy mezi entitami jsou realizovány pomocí anotací *@OneToOne*, *@OneToMany*, *@ManyToOne* a *@ManyToMany*. Tyto vztahy mohou být jednosměrné či obousměrné.

Entity jsou spravovány entity managerem, který disponuje sadou metod pro jejich vkládání, mazání a získávaní z relační databáze. Entity manager je reprezentován instancí třídy *javax.persistence.EntityManager*. Abychom tuto instanci získali, musí mít entity manager anotaci *@PersistenceContext*.

Pro získávání dat z relační databáze disponuje JPA jazykem JPQL[8] a tzv. Criteria API. JPQL je jednoduchý dotazovací jazyk svou syntaxi velmí podobný jazyku SQL. Jeho dotazy jsou tak snadno čitelné, což neplatí pro Criteria API. Výhodou Criteria API je zase typová bezpečnost jeho dotazů.

3.2.2 Enterprise Java Beans

Enterprise Java Beans, zkráceně nazývané EJB, jsou serverové komponenty běžící v EJB kontejneru, které zapouzdřují business logiku aplikace. Cílem EJB je oddělení business logiky od persistentní a prezentační vrstvy aplikace a také umožnit integraci jiných technologií. Dalším přínosem EJB je možnost vytvářet distribuované aplikace, kdy se jednotlivé EJB komponenty nacházejí na různých strojích, a také jejich schopnost řídit transakce. Enterprise beans se děli na session beans a message driven beans.

Session beans

Session beans se dělí na:

- **Stateful session beans** – Jsou stavové beany, které si uchovávají stav mezi jednotlivými požadavky klienta. Pro každého jedinečného klienta je na serveru vytvořena jemu určená beana a ta obsluhuje všechny jeho požadavky. Tyto beany se vytváří pomocí anotace *@Stateful*.
- **Stateless session beans** – Jsou bezestavové beany definované anotací *@Stateless*. Tyto beany si mezi jednotlivými požadavky klienta neuchovávají stav. Jsou združeny v tzv. poolu a při příchodu požadavku klienta se pro jeho obsluhu vyhradí libovolná z nich. Po obsloužení požadavku je tato beana vrácena zpět do poolu.

[8] Java Persistence Query Language

- **Singleton session beans** – Tyto beany jsou vždy vytvořeny při spuštění aplikace a existují po celou dobu jejího života. Singleton beana poskytuje podobnou funkcionalitou jako bezestavová beana jen s tím rozdílem, že její instaci sdílí více klientů (často i současně). Singleton beanu, stejně jako bezestavovou, lze použít pro implementaci webové služby. Vytváří se anotací *@Singleton*.

Message driven bean

Je typ beany, který umožňuje Java EE aplikacím asynchronně zpracovávat JMS[9] zprávy nebo i jiné. Tyto zprávy mohou být poslány jakoukoliv Java EE komponentou, JMS aplikací nebo systémem, který nepoužívá Java EE technologii.

3.2.3 Servlet

Servlet je Java program, který běží v rámci webového serveru a slouží pro obsluhu klientských požadavků a vytváření odpovědí. Nejčastěji se používá ve spojení s webovými klienty, kteří posílají požadavky prostřednictvím HTTP protokolu. Proto existuje třída *javax.servlet.http.HttpServlet*, která třídu *javax.servlet.GenericServlet* pro obecný servlet rozšiřuje a poskytuje konkrétní metody pro zpracování HTTP požadavků. Pro HTTP požadavek typu GET je to metoda s názvem *doGet* apod.

Obě zmíněné třídy, z nichž lze servet vytvořit, implementují rozhraní *Servlet*. Toto rozhraní definuje tři základní metody: *init()*, *service()* a *destroy()*. Tyto metody jsou volány v rámci životního cyklu servletu v uvedeném pořadí [20].

3.2.4 Java Server Faces

Java Server Faces neboli JSF je komponentový framework pro realizaci prezentační vrstvy Java EE aplikací, který běží na straně serveru a skládá se z:

- API – Reprezentuje komponenty frameworku, spravuje jejich stav, obsluhuje události, provádí validaci na straně serveru, datové konverze, definuje navigaci mezi stránkami apod.
- Knihovny tagů – Slouží pro definování komponent na webové stránce a k jejich propojování s objekty na straně serveru.

JSF v nedávné době nahradilo technologii Java Server Pages a stalo se tak standardem pro vývoj prezentační vrstvy v Java EE aplikacích. Jednou z jeho hlavních výhod je, že umožňuje čistým způsobem oddělit prezentační a business vrstvu enterprise aplikace.
Pro fungování JSF je nezbytné definovat tzv. *FacesServlet* v souboru *web.xml* enterprise serverové aplikace.

[9] Java Messaging Services, http://cs.wikipedia.org/wiki/Java_Messaging_Services

Princip komunikace JSF komponent s business vrstvou

Komponenty JSF komunikují s business vrstvou prostřednictvím expression jazyka, zkráceně EL, který umožňuje volat metody backing bean, které realizují určitou business logiku. Přístupnost backing bean v EL výrazech prostřednictvím jména beany je zajištěna pomocí anotace *@ManagedBean*. Backing beany s touto anotaci se pak nazývají managed beany.

Managed beany jsou schopné si uchovávat svůj stav mezi jednotlivými požadavky uživatele. Tohoto je docíleno prostřednictvím definování tzv. scope, což je rozsah, v rámci nějž je stav managed beany uchován. Tyto scopy jsou popsány ve specifikaci JSR 299: Contexts and Dependency Injection for the Java EE platform (CDI), jejíž referenční implementací je Weld. CDI mimo jiné popisuje princip dependency injection, který lze realizovat anotací *@Inject*, a také CDI managed beany, které jsou stejně jako managed beany přistupné v EL výrazech. CDI managed beany se definují anotací *@Named*.

Jednoduchá ukázka volání metody, resp. getteru pro atribut world, managed beany s názvem *hello* (uvedena v příkladu 7) v EL výrazu na webové stránce je zobrazena na příkladu 8.

```
@ManagedBean
public class Hello {
	final String world = "Hello World!";
	public String getworld() {
		return world;
	}
}
```

Příklad 7: Managed beana s názvem hello

```
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
	<h:head>
		<title>Facelets Hello World</title>
	</h:head>
	<h:body>
		#{hello.world}
	</h:body>
</html>
```

Příklad 8: Webová stránka s voláním getteru managed beany s názvem hello pomocí EL

3.2.5 Facelets

Je součástí specifikace Java Server Faces. Jedná se o deklarační jazyk pro vytváření webových stránek založených na JSF, které využívají HTML šablony.

Technologie Facelets zahrnuje:

- Použití XHML pro tvorbu webových stránek.

- Podporu pro Facelets, JSF a JSTL tagy.
- Podporu pro EL jazyk.
- Šablonování pro komponenty a stránky.

Výhody Facelets:

- Prostřednictvím šablonování umožňuje znovupoužití kódu.
- Umožňuje rychlejší kompilaci webových stránek.
- Validace EL výrazů při kompilaci.
- Urychluje zobrazení webových stránek.

3.2.6 PrimeFaces

PrimeFaces je open-source knihovna komponent (tzv. widgetů) pro Java Server Faces 2.0. Těchto komponent obsahuje PrimeFaces více než 100. Jsou to např. komponenty pro tvorbu kalendáře, interaktivních grafů apod. Tyto komponenty mají vestavěnou AJAX[10] funkcionalitu a jsou skinovatelné. K dalším výhodám PrimeFaces patří rozsáhlá uživatelská příručka a aktivní komunita.

Knihovna PrimeFaces je v současné době mezi vývojáři velmi populární. Svědčí o tom zveřejněný žebříček popularity obdobných knihoven, kde se Primefaces umístilo spolu knihovnou Grails na prvním místě (zjištěno z [21]).

Pro použití PrimeFaces je nutné do aplikace založené na Mavenu, resp. do jejího *pom.xml* souboru, doplnit vlastní repozitář a také závislost pro tuto knihovnu. PrimeFaces pro svůj chod vyžaduje JRE ve verzi 5+ a implementaci JSF ve verzi 2.x. Další knihovny jsou potřeba jen pro zajištění specifických funkcí knihovny. Více lze o PrimeFaces nalézt v její uživatelské příručce [22].

3.2.7 Architektura typu REST

3.2.7.1 Základní princip RESTu

REST staví na existenci zdrojů, z nichž každý lze adresovat pomocí unikátního identifikátoru (např. URL adresa v protokolu HTTP). Zdroj může mít různou reprezentaci (HTML, XML, JSON, PDF apod.) a přistupuje se k němu prostřednictvím standardizovaného rozhraní. Pomocí CRUD (create, read, update, delete) operací se pak se zdrojem manipuluje.

RESTful webová služba pro komunikaci využívá protokol HTTP a CRUD operace realizuje užitím jeho metod. Samotná přenášená data pak mohou být v různých formátech (XML, JSON, apod.) [23].

[10] Asynchronous JavaScript and XML, http://cs.wikipedia.org/wiki/AJAX

Význam HTTP metod pro konkrétní zdroj (prvek kolekce) s adresou např. *http://resource.com/12*:

- **PUT** – uložení či nahrazení prvku s ID = 12,
- **GET** – ziskani prvku dle formátu MIME[11],
- **POST** – reprezentace daného prvku jako kolekce a vložení jejího prvku,
- **DELETE** – smazání daného prvku.

Pro zdroj s adresou např. *http://resource.com/* je sémantika HTTP metod stejná, jen se pracuje s celou kolekcí prvků.

3.2.7.2 JAX-RS: Java API for RESTful Web Services

Jak již bylo řečeno, RESTful webové služby jsou realizovány pomocí JAX-RS, což je API programovacího jazyka Java, které usnadňuje jejich vývoj. JAX-RS za tímto účelem užívá anotace z balíčku *javax.wx.rs*.

Od verze 1.1 je pak JAX-RS již oficiální součástí Java EE 6, díky čemuž není pro jeho použití v tomto prostředí nutná žádná konfigurace (v jiném prostředí než Java EE 6 je potřeba malý zásah v souboru *web.xml*).

3.2.7.3 Implementace RESTful webových služeb

Webovou službu lze snadno vytvořit anotací *@Path* u obyčejné javovské třídy, která splňuje tyto podmínky:

- třída nesmí být final nebo abstract,
- musí mít implicitní bezparametrický konstruktor,
- její metody musí být public (nikoliv static nebo final).

Parametrem anotace *@Path* je URI zdroje. Tuto anotaci lze také použít u metod třídy tvořící webovou službu. V takovém případě vznikne URI zdroje konkatenací parametrů obou anotací. Metody pracující se zdrojem musí být anotovány některou z těchto anotací: *@PUT, @GET, @POST* nebo *@DELETE*. Díky příslušné anotaci u metody dojde k jejímu svázání s analogickou metodou protokolu HTTP (např. metoda třídy s anotací *@GET* bude obsluhovat HTTP požadavek typu GET apod.).

Pro specifikaci typu předávaných dat jsou užity anotace *@Produces* a *@Consumes*, jejichž parametrem je typ dat dle standardu MIME. Anotace *@Produces* se uvádí u metod s anotací *@GET*, *@POST* a *@PUT*. Anotace *@Consumes* pak u metod s anotací *@PUT* a *@POST*.

Protože HTTP požadavky běžně obsahují parametry, lze je zpřístupnit pomocí anotací *@PathParam, @QueryParam, @HeaderParam, @CookieParam, @MatrixParam* a *@Form param*,

[11] Multipurpose Internet Mail Extensions, http://cs.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions

které se uvádějí před parametry metod. Ukázka implementace webové služby s využitím anotace *@PathParam* je uvedena na příkladu 9.

```
@Path("/users/{username}")
public class UserResource {
	private String userName;

	@PUT
	@Consumes("text/xml")
	public String getUser(@PathParam("username") String userName) {
		this.userName = userName;
	}
}
```

Příklad 9:Implementace webové služby zpřístupňující parametr HTTP dotazu pomocí anotace @PathParam

Funkčnost uvedené webové služby s názvem *getUser* je taková že, pokud bude vyslán HTTP požadavek PUT na URL adresu *http://www.resource.com/users/Galileo* a tato webová služba bude umístěna na počítači s URL adresou *http://www.resouce.com* a bude aktivovaná, uloží řetězec *Galileo* do atributu s názvem *userName* třídy *UserResource*.

Více informací pro vývoj REST webových služeb lze nalézt např. v uživatelské příručce Jersey [24], což je referenční implementace JAX-RS.

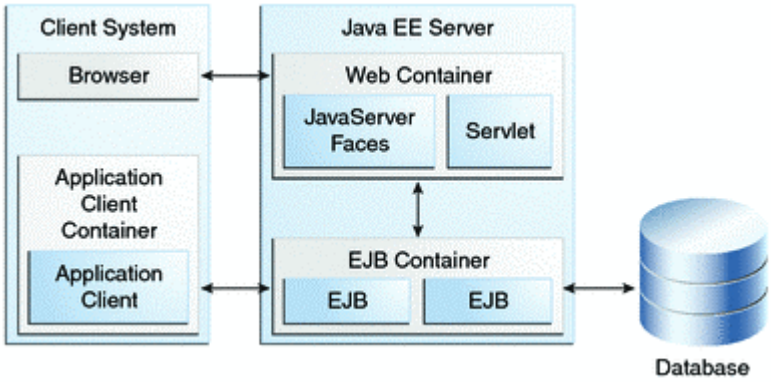
3.2.8 Aplikační server

Aplikační server vytváří prostředí pro běh enterprise Java aplikací. Skládá se z webového a EJB kontejneru. Kontejner je část serveru (virtuální prostředí), která řídí životní cyklus nasazených entit a poskytuje jim nejrůznější služby.

Webový kontejner zajišťuje chod prezentační vrstvy a obsahuje JSP, Servlety, JSF apod. EJB kontejner pak spravuje EJB komponenty (session beany a message driven beany) a umožňuje fungování business logiky aplikace tím, že řeší následující problematiku:

- **komunikace se vzdáleným klientem** – zjednodušuje komunikaci mezi klientem a aplikací,
- **dependency injection** - zajišťuje naplnění deklarovaných proměnných (datových atributů),
- **řízení stavu** - kontejner udržuje v paměti stavy jednotlivých stavových (stateful) beanů a tím i vzdálený stav u klienta, kterému se jeví stav jakoby uložen lokálně,
- **pooling** - vytváření poolu instancí pro bezstavové a message driven beany,
- **řízení životního cyklu** - stará se o vytváření, inicializaci a destrukci instancí beanů a další události,
- **management transakcí** – beany deklarují transakční vlastnosti metod, kontejner řeší commit a rollback,
- **bezpečnost** – deklarace přístupů na úrovni tříd a metod.

Schéma znázorňující aplikační server spolu s tenkým a tlustým klientem a použitou databází je na obrázku 2.



Obrázek 2: Schéma aplikačního serveru, převzato z [25]

Aplikačních serverů existuje celá řada. Některé plně podporují standard Java EE platformy a jiné jen z části. O tom, zda aplikační server podporuje standard Java EE platformy, rozhoduje sada testů kompatibility společnosti Sun Microsystems.

Aplikační servery lze také rozdělit podle toho, zda jsou open-source čí komerční. Mezi open-source aplikační servery patří např. GlassFish, JBoss, JonAs, Apache Geronimo a Apache Tomcat (částečná implementace). Zástupci komerčních aplikačních serveru jsou např. IBM Wepshere, BEA WebLogic a Sun Java System Application ServerOracle AS [26].

JBoss Application Server

Vzhledem k použití nástroje JBoss Tools se jako logické ukázalo použít JBoss Application Server (dále jen JBoss AS), resp. jeho aktuálně nejnovější verzi JBoss AS 7.1. Tento server plně podporuje standard Java EE platformy. Jeho dokumentace je dostupná zde [27].

3.2.9 JBoss AS7 Deployment Plugin

Tento plugin umožňuje jediným příkazem nasadit JEE aplikaci na JBoss AS (*mvn jboss-as:deploy*), opětovně nasadit (*mvn jboss-as:redeploy*), či ji odstranit ze serveru (*mvn jboss-as:undeploy*). Analogicky plugin podporuje také nasazování artefaktů (např. JDBC ovladač apod.) nebo přidávání, resp. odebírání, zdrojů (resources), ale pro tento účel nebyl v práci použit. Další informace o pluginu lze nalézt v [28].

3.2.10 PostgreSQL

Jako databázi pro uložení dat serverové aplikace jsem zvolil PostgreSQL. Jedná se o open-source objektově-relační databázový systém, který vyniká svou stabilitou a rychlostí.

Podporuje normy SQL92 a SQL99 a mnoho dalších moderních rysů: funkce, indexy, triggery, uložené procedury, transakce, MVCC (řeší současný přistup k databázi), pravidla, mnoho datových typů, uživatelem definované datové typy, dědičnost a mnoho dalších (viz. např. [29]).

3.2.11 JBoss Tools

JBoss Tools je souhrnný název pro sadu Eclipse pluginů vyvíjenou divizí JBoss společnosti Red Hat, které usnadňují práci s JBoss produkty jako Hibernate, JBoss AS, OpenShift, Drools, jBPM, Seam, JBoss ESB, JBoss Portal a další. Mimo tyto technologie JBoss Tools poskytuje podporu také pro Maven, JSF, (X)HTML apod. [30]

3.2.12 OpenShift

OpenShift je cloudové řešení typu PaaS (Platform as a Service) od firmy Red Hat. Jeho veřejná verze poskytuje po registraci prostředí pro nasazení a provoz různých typů aplikací. Podporovány jsou aplikace implementované v jazycích Java, PHP, Ruby a Python.

Před samotným nasazením uživatelské aplikace je nutno nejprve vytvořit tzv. stack, který definuje platformu pro tuto aplikaci (použitý server databázi apod.). Stack lze vytvořit prostřednictvím webového rozhraní nebo prostřednictvím nástroje *rhc* z příkazové řadky linuxu.

Aby mohla být aplikace nasazena na OpenShift, musí být verzována pomocí verzovacího nástroje GIT[12]. Aplikaci pak lze nasadit přímo s jeho užitím z příkazové řádky nebo prostřednictvím IDE s podporou OpenShiftu [31].

[12] http://git-scm.com

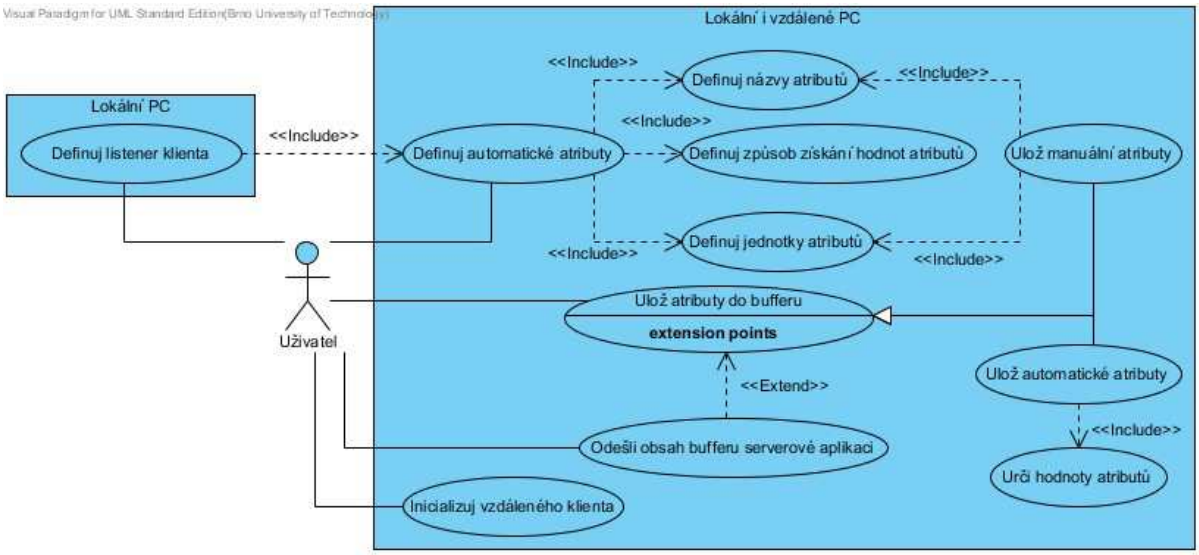
4 Realizace klientské části aplikace

V této kapitole práce je popsána realizace klientské části aplikace od fáze analýzy až po její implementaci. Nejprve je představena analýza a návrh klientské aplikace a jejích komponent a poté je popsána implementace klientské části aplikace jako celku. Při realizaci klientské části aplikace se vycházelo z požadavků zadavatele, jejichž specifikace je uvedena v příloze A

4.1 Analýza a návrh klientské aplikace

4.1.1 Případy užití klienta

Na základě analýzy specifikace požadavků byly u klienta identifikovány případy užití, které jsou zobrazeny na obrázku 3. Tyto případy užití zachycují použití klienta na lokálním, resp. testovacím, počítači a na vzdáleném, resp. testovaném počítači, které musí být ze strany klienta podporovány proto, aby se klienti mohli plně účastnit distribuovaného testování platformy JBoss.



Obrázek 3: Diagram případů užití klientské aplikace

Popis hlavních případů užití klienta:

Definuj listener klienta

Uživatel definuje listener, který bude klient používat v jednotkových testech, pro získávání požadovaných výkonnostních dat. Definice listeneru mimo jiné zahrnuje i jeho specifikaci prostřednictvím Maven Surefire Pluginu v souboru *pom.xml* testovaného projektu.

Definuj automatické atributy

Automatické atributy mohou být zjišťovány kdekoliv uvnitř testu, ale implicitně mají být měřeny v rámci listeneru klienta (podrobněji v podkapitole 4.1.5). Automatické atributy budou muset být proto definovány uvnitř konfigurační třídy, která se předá listeneru klienta ještě před zahájením jeho činnost. Definování každého atributu zahrnuje specifikaci jeho názvu, funkce (získává hodnotu atributu a čas, kdy byla hodnota změřena) a jednotky, v níž budou atributy měřeny.

Ulož atributy do bufferu

Jedná se o obecný případ pro uložení automaticky získaných nebo manuálně zjištěných výkonnostních atributů do bufferu klienta (vysvětlen v podkapitole 4.1.3). Uložení automatických atributů do bufferu klienta zahrnuje vždy změření hodnot všech těchto definovaných atributů a to včetně určení času jednotlivých měření. Uložení každého manuálního atributu do bufferu klienta zahrnuje definici jeho názvu, jednotky a jeho hodnoty (zjištěna z vnějšku klienta). Při vlastním uložení atributu se také stanoví čas jeho měření. Tento případ užití může být ještě rozšířen o odeslání obsahu bufferu klienta serverové aplikaci.

Odešli obsah bufferu serverové aplikaci

Odešle aktuální obsah bufferu klienta s výkonnostními daty serverové aplikaci.

Inicializuj vzdáleného klienta

Slouží pro předání inicializačních dat směrem od lokálního klienta ke vzdálenému, případně od vzdáleného klienta ke vzdálenému tak, aby vzdálený inicializovaný klient mohl odesílat svá specifická výkonnostní data serverové aplikaci a tato data byla uložena v rámci odpovídající testované metody.

4.1.2 Módy klienta

Pro rozlišení činnosti klienta na lokálním a vzdáleném počítači byl zaveden tzv. lokální a vzdálený mód klienta, přičemž v obou těchto módech budou moci klienti měřit automatické a manuální výkonnostní atributy (viz. obrázek 3).

Lokální mód

Klient pracuje v tomto módu na počítači, na kterém probíhá jednotkové testování. Při tomto testování využívá vlastní implementaci některého z listenerů. Kompletní nastavení testu (včetně URL adresy, na kterou bude posílat data) je klientovi předáno při vytváření instance použitého listeneru. V této fázi se mu také předloží definice automaticky měřených atributů. Klient v tomto módu provádí výkonnostní měření ve všech testech s výjimkou těch, které byly z testování explicitně vyloučeny.

Vzdálený mód

V tomto módu pracují klienti na vzdálených počítačích (vzdálení klienti). Ti musí být pro svou činnost inicializováni lokálním klientem (předá URL serverové aplikace a ID test suity). Definice automaticky měřených atributů je klientovi v tomto módu předána prostřednictvím jiné metody, než je tomu u lokálního klienta. Tato metoda také určí, že klient bude pracovat jako vzdálený a jako URL adresu serverové aplikace bude uvažovat tu, kterou mu předal lokální klient. Vzdálení klienti se účastní pouze distribuovaných testů.

4.1.3 Paměť klienta

Paměť klienta neboli buffer bude uchovávat všechna naměřená výkonnostní data. Buffer bude vhodné implementovat jako hashovací tabulku, jejímž klíčem bude název měřeného výkonnostního atributu a hodnotou pak objekt, který bude obsahovat jednotku, v níž se atribut měří, a dále kolekcí naměřených hodnot. Každá hodnota z kolekce bude reprezentována dvojicí čas a vlastní naměřená hodnota.

Tato reprezentace paměti umožní snadné přidávání naměřených výkonnostních dat a to tak, že se v tabulce vyhledá objekt odpovídající měřenému výkonnostnímu atributu a do jeho kolekce se přidá vlastní naměřená hodnota tohoto atributu. Při každém dokončeném testu se pak stávající obsah bufferu odešle serverové aplikaci a současně se vyprázdní.

4.1.4 API klienta

API klienta již bylo částečně nastíněno v podkapitole 4.1.1. V této formě by však jeho použití bylo uživatelsky nepříliš přívětivé a pro reálné získávání a odesílání výkonnostních dat málo flexibilní. Aby tedy metody API poskytovaly uživateli co největší svobodu při měření výkonnostních dat, měla by existovat samostatná metoda pro získání automaticky měřených atributů, pro uložení hodnot manuálně získaných atributů, ale také metoda, která funkčně pokrývá obě předchozí metody. Pro zmíněné metody by měla také existovat jejich varianta s následným odesláním bufferu. Je vhodné, aby metoda pro odeslání bufferu klienta existovala také samostatně.

Vedle zmíněných metod bude nezbytná také metoda pro vytvoření instance klienta v lokálním módu (konstruktor), dále metoda pro přepnutí instance klienta do vzdáleného módu včetně případného předání konfigurační třídy obsahující definované automaticky měřené atributy a metoda pro inicializaci vzdáleného klienta.

4.1.5 Listenery klienta

Klient v lokálním módu má při spuštění a na konci každého jednotkového JUnit či TestNG testu změřit automatické výkonnostní atributy. Současně má změřit také metodou spotřebovaný procesorový čas a JVM paměť, resp. její změnu, a to bez toho, aby byl v testu uživatelem

instanciován a volán. Pro dosažení této funkcionality je nutno, aby disponoval vlastní implementací listenerů těchto knihoven[13].

Zjednodušený návrh možné implementace JUnit a TestNG listeneru klienta, realizující požadovanou funkčnost i v návaznosti na paměť klienta a jeho komunikaci se serverovou aplikací, je představen v příkladech v podkapitolách 4.1.5.1 a 4.1.5.2. Uvedené příklady používají následující metody:

- **getCpuTime()** – Metoda zjistí aktuální procesorový čas.
- **getJVM()** – Metoda zjistí aktuálního využití JVM paměti.
- **gainAutoAttributesAndSaveThemToBuffer()** – Metoda určí hodnoty definovaných automatických výkonnostních atributů a uloží je do bufferu klienta.
- **add2ValuesToBuffer(value1, value2)** – Uloží dvě hodnoty value1 a value2 do bufferu klienta.
- **sendBufferToServer()** – Odešle obsah bufferu serverové aplikaci.

4.1.5.1 JUnit listener klienta

Na příkladu 10 je ukázán návrh implementace JUnit listeneru klienta. Třída listeneru dědí ze třídy *RunListener* a předefinovává její metody s názvem *testStarted* a *testFinished*.

```
public class ClientJUnitListener extends RunListener  {
	private long cpuTime, jvm;
	// Metoda se volá při spuštění testu.
	public void testStarted(Description description) {
		cpuTime = getCpuTime();
		jvm = getJVM();
		gainAutoAttributesAndSaveThemToBuffer();
		sendBufferToServer();
	}
	// Metoda se volá po skončení testu.
	public void testFinished(Description description) {
		add2ValuesToBuffer(getCpuTime() - cpuTime, getJVM() - jvm);
		gainAutoAttributesAndSaveThemToBuffer();
		sendBufferToServer();
	}
}
```

Příklad 10: Zjednodušený návrh implementace JUnit listeneru klienta pro měření výkonnostních dat

4.1.5.2 TestNG listener klienta

Na příkladu 11 je ukázán návrh implementace TestNG listeneru klienta. Třída listeneru rozšiřuje rozhraní *ITestListener* a implementuje jeho metody s názvem *onTestStart, onTestSuccess* a *onTestFailure*.

[13] Listener pro JUnit byl představen v podkapitole 2.1.1 a pro TestNG v podkapitole 2.1.2

```
public class ClientTestNGListener implements ITestListener {
	private long cpuTime, jvm;
	// Metoda se volá při spuštění testu.
	public void onTestStart(ITestResult result) {
		cpuTime = getCpuTime();
		jvm = getJVM();
		gainAutoAttributesAndSaveThemToBuffer();
		sendBufferToServer();
	}
	// Metoda se volá u úspěšného testu.
	public void onTestSuccess(ITestResult result) {
		handleFinishedTest(result);
	}
	// Metoda se volá u neúspěšného testu.
	public void onTestFailure(ITestResult result) {
		handleFinishedTest(result);
	}
	// Obslužná metoda pro konec testu.
	private void handleFinishedTest(ITestResult result) {
		add2ValuesToBuffer(getCpuTime() - cpuTime, getJVM() - jvm);
		gainAutoAttributesAndSaveThemToBuffer();
		sendBufferToServer();
	}
	// Prázdné implementace ostatních metod rozhraní ItestListener.
}
```

Příklad 11: Zjednodušený návrh implementace TestNG listeneru klienta pro měření výkonnostních dat

4.1.6 Komunikace klienta s ostatními entitami

Pro komunikaci klienta se serverovou aplikací a ostatními klienty bude muset disponovat třídou pro navázání TCP spojení a posílání HTTP požadavků. Proč byl pro komunikaci zvolen protokol HTTP a jak je komunikace realizována, je podrobně vysvětleno v kapitole 6.

4.1.7 Inicializace vzdáleného klienta a princip řešení jeho komunikace

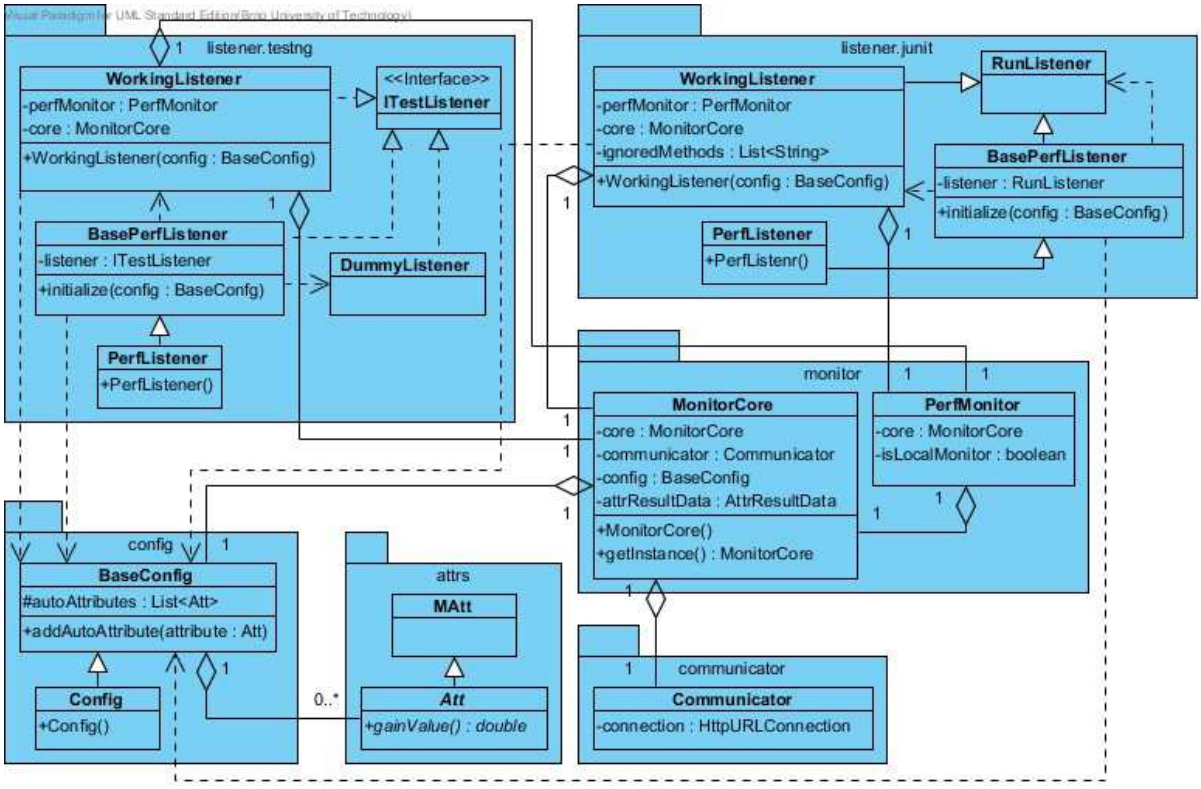
Vzdálený klient nemá k dispozici počáteční konfiguraci testovaného projektu (především tedy URL adresu serverové aplikace), která byla předána lokálnímu klientovi. Lokální klient mu tedy musí poskytnout inicializační data. Vzhledem k tomu, že při testování může být testováno současně několik test suit v určitém projektu, vzdálený klient bude muset znát také ID test suity, na jejímž testování se právě podílí. Toto ID nejprve určí lokální klient a teprve poté bude moci inicializovat vzdáleného klienta(y). Inicializaci bude vhodné provést v metodách s anotacemi *@BeforeClass* (u JUnit) případně *@BeforeSuite* (u TestNG) ve zdrojových kódech testů, čímž se zajistí včasná inicializace vzdáleného klienta.

Vlastní inicializace vzdáleného klienta bude provedena vysláním HTTP požadavku typu GET s danými inicializačními daty na URL adresu testované enterprise aplikace. Tato data předaná prostřednictvím URL pak zpracuje inicializační servlet (popsán v podkapitole 4.2) a inicializuje tak vzdáleného klienta. Ten pak bude moci provádět vlastní měření a odesílání výkonnostních dat serverové aplikaci.

Zahájení měření vzdáleného klienta je však podmíněno nějakou interakcí lokálního klienta s testovanou enterprise aplikací (např. vysláním HTTP požadavku typu GET pro získání určité webové stránky uvažované enterprise aplikace).

4.1.8 Návrh struktury klienta

Na základě předchozí analýzy všech klíčových oblastí klienta, týkajících se jeho způsobilosti účastnit se testování platformy JBoss, byl vytvořen jeho počáteční návrh (obrázek 4).



Obrázek 4: Návrhový diagram tříd klienta

Význam jednotlivých balíčku na obrázku 4:

attrs

Tento balíček sdružuje třídy pro automatické a manuální výkonnostní atributy. Třída *MAtt* reprezentuje manuální atribut, který nese název, jednotku, vlastní hodnotu a čas, kdy byla tato

hodnota získána. Abstraktní třída *Att* (dědí z třídy *MAtt*) představuje automatický atribut. Pro jeho vytvoření je potřeba vytvořit instanci této třídy a implementovat její abstraktní metodu *public double gainValue()*, která hodnotu atributu získá (metoda implicitně měří i čas, kdy byla hodnota atributu získána).

config

Balíček *config* obsahuje konfigurační třídy pro listenery klienta. Třída *BaseConfig* obsahuje základní specifika o testovaném projektu (název projektu, buildu, test suity a testovací platformu) a v atributu s názvem *autoAttributes* uchovává seznam všech automaticky měřených výkonnostních atributů. Třída *Config*, která ze třídy *BaseConfig* dědí, bude tento seznam atributů ve svém konstruktoru inicializovat voláním metody *public void addAutoAttribute(Att attribute).*

listener.junit

Balíček obsahuje třídy, které realizují požadovanou funkčnost JUnit listeneru klienta. Význam tříd je následující:

- **RunListener** – Třída byla představena v podkapitole 2.1.1.
- **WorkingListener** – Potomek třídy *RunListener* realizující měření výkonnostních dat. Při jeho implementaci bude použit návrh pro JUnit listener z podkapitoly 4.1.5.1.
- **BasePerfListener** – Je potomek třídy *RunListener*, který v případě chybné konfigurace klienta využívá instanci třídy *RunListener* (standardní JUnit listener), jinak instanci třídy *WorkingListener*.
- **PerfListener** – Standardní listener s předdefinovanými automatickými atributy, který klient používá pro JUnit testy.

listener.testng

Balíček obsahuje třídy, které realizují požadovanou funkčnost TestNG listeneru klienta. Význam tříd *WorkingListener*, *BasePerfListener* a *PerfListener* je analogický stejnojmenným třídám z balíčku *listener.junit.* Rozhraní *ITestListener* bylo představeno v podkapitole 2.1.2 a třída *DummyListener* je prázdnou implementací rozhraní *ITestListener* (neprovádí tedy žádnou činnost). Pro implementaci třídy *WorkingListener* z tohotu balíčku byl využit návrhu TestNG listeneru z podkapitoly 4.1.5.2.

communicator

Tento balíček obsahuje třídu *Communicator*, která slouží pro komunikaci klienta se serverovou aplikací, případně se vzdálenými klienty. Třída využívá pro samotnou komunikaci objekt třídy *HttpURLConnection*, který umožňuje vytvořit TCP spojeni, a metody, které prostřednictvím tohoto spojení budou schopny odesílat a přijímat HTTP zprávy.

monitor

V tomto balíčku jsou třídy, které tvoří jádro klienta. Třída *MonitorCore* zapouzdřuje objekt třídy *Communicator* a samotný buffer klienta (objekt třídy *AttrResultData*). Tato třída je navržena dle návrhového zdroje singleton a je sdílena listenery klienta a třídou *PerfMonitor*. Metody třídy *PerfMonitor* tvoří API klienta a její atribut s názvem *isLocalMonitor* rozlišuje jeho mód.

4.2 Analýza a návrh komponent klientské aplikace

Na základě analýzy specifikace požadavků klientské části aplikace a předešlé analýzy a návrhu klienta byly pro požadovanou funkčnost klientské části aplikace identifikovány níže uvedené komponenty. Ty budou realizovány jako samostatné jednotky, čímž se zajistí jejich snadná integrovatelnost do testované enterprise aplikace.

Inicializační servlet

Slouží pro získávání inicializačních dat a jejich předávání vzdálenému klientovi. Servlet zpracovává přijaté HTTP požadavky typu GET a získává z nich URL adresu serverové aplikace a ID test suity. Tyto hodnoty poté ukládá do systémových proměnných. Tímto způsobem budou daná data dostupná pro vzdáleného klienta až do doby, než servlet zpracuje jiný příchozí požadavek s inicializačními daty.

Nástroj pro měření response time

Nástroj pro měření response time (odezvy webového serveru) bude vytvořen s použitím aplikace Response-Time Filter[14]. Implementace tohoto nástroje bude provedena tak, že klient bude začleněn do aplikace Response-Time Filter. Ta bude modifikována tak, aby prováděla pouze měření doby odezvy. Po jejím určení pak klient, který bude ve vzdáleném módu, tuto hodnotu (manuální výkonnostní atribut) prostřednictvím metody svého API odešle serverové aplikaci.

4.3 Implementace klientské části aplikace

Implementace klientské části aplikace zahrnuje implementaci klienta a jeho komponent. Všechny tyto komponenty byly naprogramovány v jazyce Java jako Maven projekty.

[14] Představen v podkapitole 3.1.5.

4.3.1 Realizace klienta

Klient byl realizován jako projekt s názvem *PerfClient* podle jeho navržené struktury v podkapitole 4.1.8, přičemž listenery z tohoho návrhu bylo nutné ještě rozšířit o určení názvu testované metody (včetně jejího ballíčku), zjištění obsahu případné výjimky při testu a v případě TestNG listeneru také o určení parametrů, se kterými se metoda testovala (JUnit parametry zjistit bohužel neumožňuje). Tato data byla v případě TestNG listeneru zjišťována z objektu s rozhraním *ITestResult* a v případě JUnit listeneru z objektů tříd *Description* a *Failure* (zapouzdřuje obsah výjimky).

Metody, které jsou při testování ignorovány (v JUnit pomoci anotace *@Ignore*, v TestNG pomocí nastavení vlastnosti testu *enabled* na hodnotu *false*), jsou listenery klienta odesílány serverové aplikaci až při dokončení testování test suity. Je to z toho důvodu, že ačkoliv by JUnit listener jejich zasílání umožňoval okamžitě, TestNG listener má tyto metody dostupné až při dokončení testování test suity v metodě s názvem *onFinish* (konkrétně v jejím parametru s rozhraním *ITestContext*). Proto se k zasílání ignorovaných metod přistoupilo již zmíněným způsobem, čímž se chování obou listenerů ujednotilo.

Paměť klienta je realizována dle jejího návrhu v podkapitole 4.1.3. Konkrétně ji tvoří třída s názvem *AttrResultData*, která byla současně použita pro přenos samotných výkonnostních dat mezi klientem a serverovou aplikací. Tato třída byla implementována v rámci Maven projektu s názvem *PerfObjects*, který obsahuje všechny třídy používané pro přenos dat mezi klientem a serverovou aplikací (struktura těchto tříd i jejich význam je popsán v podkapitole 6.2.1). Mimo tyto třídy obsahuje projekt *PerfObjects* také sdílené konstanty oběma aplikacemi (URL adresy webových služeb apod.). Při implementaci klienta byl tento projekt pomocí Maven závislosti vložen do *pom.xml* souboru klienta a všechny jeho třídy a konstanty tak byly pro implementaci klienta zpřístupněny (analogicky byl projekt *PerfObjects* použit také při implementaci serverové aplikace).

Klient získává výkonnostní data (s výjimkou JVM paměti) s pomocí knihovny SIGAR a svou činnost loguje s využitím knihoven SLF4J a LOG4J, kdy zaznamenává informace o zaslaných datech serverové aplikaci spolu s přijatými HTTP kódy, které mu serverová aplikace vrací[15].

Výsledné API klienta bylo realizováno dle jeho návrhu v podkapitole 4.1.4. Toto API je tvořeno ve finální implementaci klienta všemi veřejnými metodami třídy *PerfMonitor*. Finální diagram tříd klienta, který odpovídá jeho implementaci, lze nalézt v příloze B.

4.3.2 Realizace komponent klienta

Inicializační servlet

Inicializační servlet byl pro vzdáleného klienta implementován v projektu s názvem *PerfClientServlet* dle navrženého řešení (uvedeno v podkapitole 4.2). Tento projekt obsahuje jedinou třídu (servlet)

[15] Ukázka reálného logu klientovy činnosti je na přiloženém DVD.

s názvem *PerfClientInitServlet*, jejíž zjednodušená implementace je znázorněna na příkladu 12. Tento servlet z přijatého HTTP požadavku typu GET získá hodnoty parametrů *repurl* (adresa serverové aplikace) a *testsuiterunid* (ID testované test suity) a uloží je do svých lokálních proměnných s názvem *repurl* a *testSuiteRunId*.

```
public class PerfClientInitServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
	public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
                                                    ServletException, IOException {
		String repURL = req.getParameter("repurl");
		String testSuiteRunId = req.getParameter("testsuiterunid");
		System.setProperty("repURL", repURL);
		System.setProperty("testSuiteRunId", testSuiteRunId);
		// Kód pro generování informační jsp stránky.
	}
}
```

Příklad 12: Zjednodušená implementace inicializačního servletu pro vzdáleného klienta

Nástroj pro měření response time

Tento nástroj byl vytvořen s užitím zdrojových kódů aplikace Response-Time Filter, kdy jeho třída s názvem *RtFilter* byla zbavena nepotřebného kódu a její metoda s názvem *doFilter* byla implementována dle příkladu 13.

Instance vzdáleného klienta *p*, která je v příkladu používána, je vytvořena příkazem *new PerfMonitor().asRemote();*. Po zjištění času *t2*, je pak do seznamu manuálních atributů *attrs* vložen atribut pro response time se zjištěnou dobou odezvy (*t2 – t1*). Tento atribut je pak s pomocí metody klienta s názvem *sendAttributes* odeslán serverové aplikaci (klient však musí být nejprve inicializován pomocí inicializačního servletu). Projekt s nástrojem pro měření response time nese název *PerfRtFilter*.

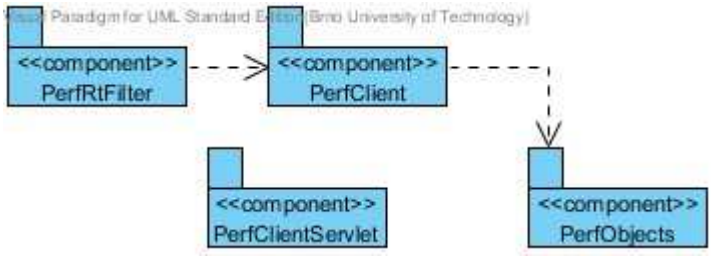
```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain)
                    throws IOException, ServletException {

       long t1 = 0;
       RtFilterResponseWrapper hresp = new RtFilterResponseWrapper(resp);
       synchronized (lock) {
              t1 = System.currentTimeMillis();
       }
       try { chain.doFilter(req, hresp);
       }
       finally {
              synchronized (lock) {
              long t2 = System.currentTimeMillis();
              // Vytvoření seznam atributů s jednou položkou manuálního atributu
              // pro response time.
              MAtt[] attrs = {new MAtt(A.REMOTE_RESPONSE_TIME, U.MILLISEC,
                                          (double) (t2 - t1)) {}};
              // Odeslání položky atributu v seznamu attrs serverové aplikaci.
              p.sendAttributes(attrs);
       }
}
```

Příklad 13: Implementace metody doFilter() třídy RtFilter.java pro měření a odesílání response time

4.3.3 Výsledná struktura klientské části aplikace

Klientskou část aplikace, resp. její finální verzi, tvoří komponenty (Maven projekty) *PerfObjects*, *PerfClient*, *PerfClientServlet* a *PerfRtFilter*. Tyto komponenty jsou včetně jejich vzájemných závislostí zobrazeny na obrázku 5.



Obrázek 5: Diagram komponent klientské části aplikace

Zdrojové kódy všech těchto projektů (komponent) jsou umístěny na přiloženém DVD. Uživatelský manuál pro klientskou část aplikace se nachází v příloze G.

5 Realizace serverové aplikace

V této kapitole je popsána realizace serverové aplikace od fáze analýzy až po její implementaci.

5.1 Analýza a návrh

Analýza i návrh serverové aplikace vychází z požadavků zadavatele. Jejich specifikace je uvedena v příloze. Ve specifikaci požadavků nejsou detailně popsány srovnávací pohledy na výkonnostní data, a proto je tomu tak učiněno zde.

5.1.1 Srovnávací pohledy na výkonnostní data

Srovnávací pohledy na výkonnostní data jsou hlavním cílem serverové aplikace. Uživateli by měly poskytnout komplexní nástroj pro analýzu výkonnostních dat v určitém projektu dle zvolených kritérií.

Pohled typu I

Pohled umožňuje srovnání vybrané test suity, resp. jejích běhů, v rámci určitého buildu projektu a zvoleného výkonnostního atributu napříč vybranými testovacími platformami. Toto porovnání je realizováno pro každou metodu test suity zvlášť (ukázka výsledné realizace pohledu je v příloze D).

Pohled typu II

Tento pohled umožňuje srovnání dvou běhů test suit z hlediska zvolených výkonnostních atributů. Jeden běh test suity je považován za referenční a druhý je vůči němu porovnáván. U každé metody z test suity bude tedy pro daný výkonnostní atributu zobrazena jeho hodnota naměřená v obou bězích test suit, dále pak rozdíl těchto hodnot a jejich relativní porovnání. Uživatel tak snadno určí, jak si výkonnostně stojí konkrétní metoda z porovnávaného běhu test suity vůči metodě z referenčního běhu test suity.

V rámci tohoto pohledu se budou brát v potaz také hraniční hodnoty pro globální atributy (globální omezení) a atributy u projektu (lokální omezení), které může uživatel stanovit. Tyto hraniční hodnoty budou srovnávány s vypočtenými rozdíly hodnot z obou běhů test suit, a pokud bude hodnota takového rozdílu stejná či větší, bude jeho hodnota odlišena. To bude signalizovat, že došlo k překročení tolerované hodnoty a došlo tedy k nežádoucímu výkonnostnímu propadu. Při překročení obou typů hraničních hodnot bude mít vyšší prioritu překročení lokálního omezení atributu (ukázka výsledné realizace pohledu je v příloze E).

Pohled typu III

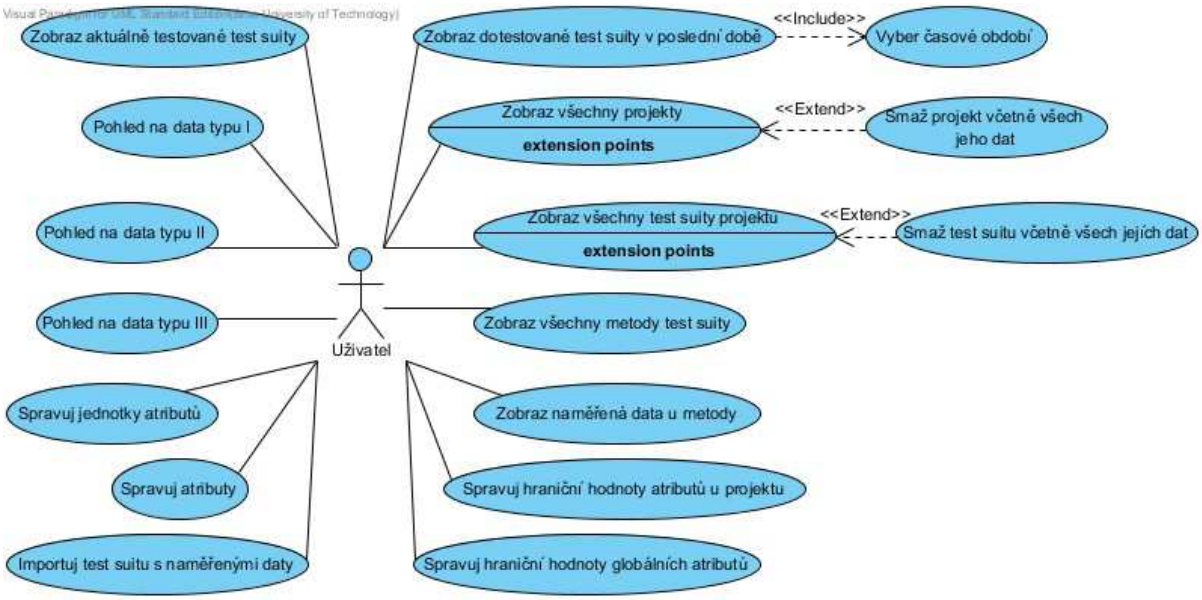
Umožňuje srovnání všech běhů konkrétní metody v rámci zvolené test suity z hlediska určitého výkonnostního atributu napříč zvolenými testovacími platformami (ukázka výsledné realizace pohledu je v příloze F).

5.1.2 Diagram případů užití

Na základě specifikace požadavků lze u serverové aplikace identifikovat aktéry uživatel a systém.

5.1.2.1 Případy užití aktéra uživatel

Diagram případů užití tohoto aktéra je znázorněn na obrázku 6.



Obrázek 6: Diagram případů užití aktéra uživatel

Popis případů užití aktéra uživatel:

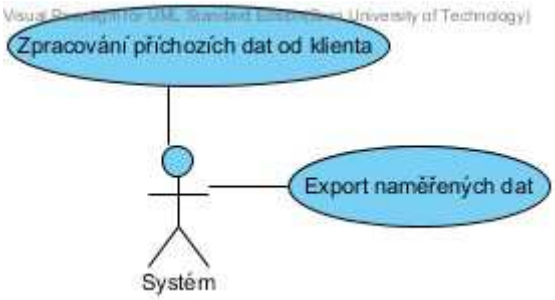
- **Spravuj jednotky atributů** – Umožní spravovat názvy jednotek pro výkonnostní atributy.
- **Spravuj atributy** – Umožní spravovat výkonnostní atributy včetně jejich jednotek, které jsou podporovány serverovou aplikací.
- **Pohled na data typu I, II, II** – Pohledy funkčně odpovídají jejich popisu v podkapitole 5.1.1. Jsou přístupné, pokud si uživatel zobrazí všechny projekty.
- **Zobraz aktuálně testované test suity** – Zobrazí testované test suity ze všech projektů
- **Zobraz dotestované test suity v poslední době** – Implicitně zobrazuje test suity ze všech projektů, které byly dotestovány v posledních deseti dnech. Umožňuje však i zvolit časové období, v rámci nějž se budou dotestované test suity zobrazovat.
- **Zobraz všechny projekty –** Zobrazí názvy všech projektů uložených test suit.

- **Zobraz všechny test suity projektu** – Zobrazí všechny test suity ze všech buildů zvoleného projektu.
- **Zobraz všechny metody test suity** – Zobrazí všechny testované metody v rámci zvolené test suity. U každé zobrazené metody bude uveden její název, zda skončila chybou či nikoliv, případná výjimka a parametry, se kterými byla při testování volána. V seznamu metod mají být také metody, které byly v průběhu testování explicitně vyloučeny.
- **Zobraz naměřená data u metody** – Umožní zobrazit data všech měřených výkonnostních atributů u zvolené metody. Data každého výkonnostního atributu budou zobrazena ve formě grafu a tabulky a budou z nich určeny základní statistické údaje (průměrná, minimální a maximální hodnota, variační rozpětí[16], směrodatná odchylka a rozptyl). Kromě těchto údajů bude na stránce zobrazen popis testované metody (pokud byl u testované metody definován) a případná výjimka, se kterou metoda skončila.
- **Spravuj hraniční hodnoty atributů u projektu** – U zvoleného projektu umožňuje k výkonnostním atributům definovat maximální tolerované hodnoty. Toto nastavení se projevuje v pohledu typu II a při exportu dat pro portál qVue. Pokud bude některá z těchto hodnot při výkonnostním srovnávání překročena hodnotou příslušného atributu, tato hodnota bude odlišena (u pohledu typu II červeně a v exportu pro portál písmenem L).
- **Spravuj hraniční hodnoty globálních atributů** – Poskytuje analogickou funkčnot jako případ použití výše jen s tím rozdílem, že tyto hraniční hodnoty jsou společné pro všechny projekty. Při překročení hraniční hodnoty atributu pří výkonnostním srovnávání je hodnota příslušného atributu u pohledu typu II zbarvena modře a v exportu pro portál qVue označena písmenem G.
- **Importuj test suitu s naměřenými daty** – Umožňuje importovat celou test suitu se všemi jejími metody a změřenými výkonnostními daty v XML souboru.

[16] rozdíl maximální a minimální hodnoty

5.1.2.2 Případy užití aktéra systém

Diagram případů užití tohoto aktéra je znázorněn na obrázku 7.



Obrázek 7: Diagram případů užití aktéra systém

Popis případů užití aktéra systém:

- **Zpracování příchozích dat od klienta** – Tento případ užití zahrnuje veškerou komunikaci s klienty při výkonnostním testování platformy JBoss.
- **Export naměřených dat –** Umožňuje ve formátu JSON předat výkonnostní data, resp. výkonnostní srovnání zvolené test suity ze dvou buildů téhož projektu v rámci všech měřených výkonnostních atributů. Vždy se porovnávají poslední dokončené test suity. Pokud jsou uvažovány dva stejné buildy, porovná se poslední dokončená test suita s test suitou, která byla dokončena hned před ní.

5.2 Implementace

Serverová aplikace byla implementována jako pěti vrstvá Java EE aplikace v Maven projektu s názvem *PerfServer*.

5.2.1 Vrstvy serverové aplikace

Vrstvy serverové aplikace včetně použitých technologií pro jejich realizaci jsou uvedeny v tabulce 3.

Vrstva aplikace	**Použité technologie**
Klientská vrstva	Webový prohlížeč
Prezentační vrstva	Facelets, JSTL, JSF, PrimeFaces
Business vrstva	DI (dependency injection), managed beans, REST API
Perzistentní vrstva	JPA, EJB
Databázová vrstva	PostgreSQL

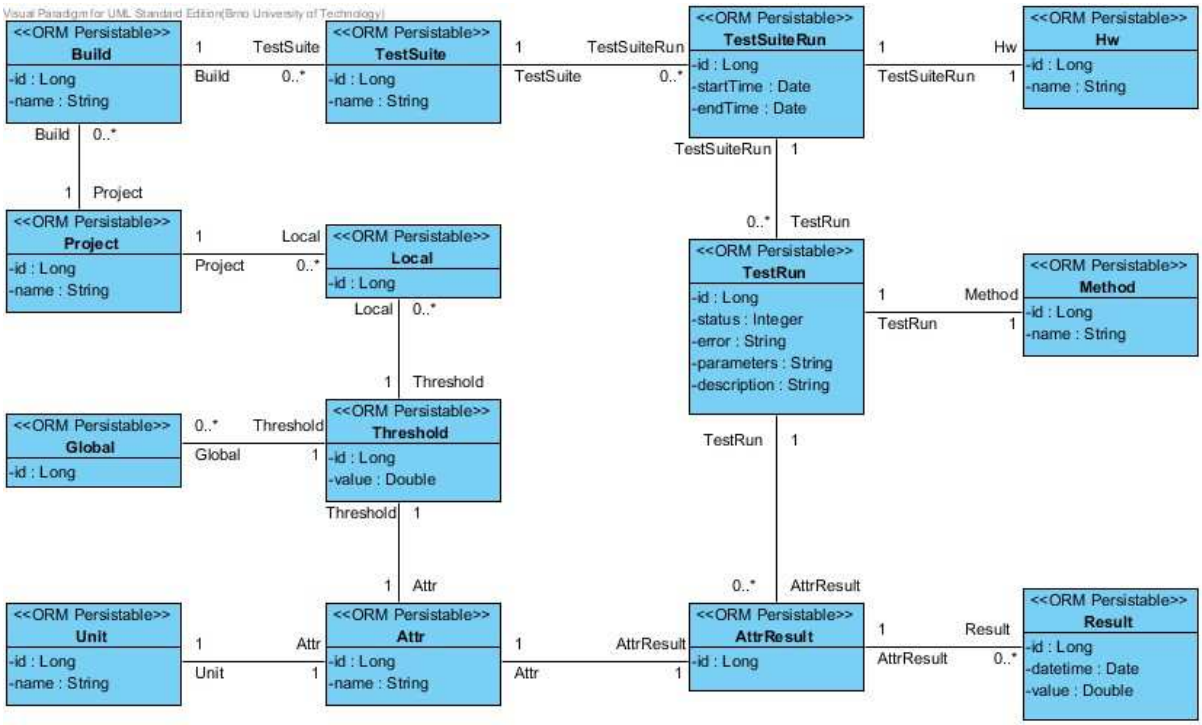
Tabulka 3: Vrstvy serverové aplikace včetně použitých technologií

5.2.2 Realizace perzistentní vrstvy

Perzistentní vrstva je realizována pomocí doménových objektů a návrhového vzoru DAO (Data Access Object), který zapouzdřuje přístup k datovým zdrojům a skrývá jejich implementaci prostřednictvím použitého rozhraní. DAO tak umožňuje, aby aplikace mohla být snadno přizpůsobena pro práci s jiným datovým zdrojem a to aniž by musely být modifikovány její business komponenty. DAO tak plní funkci adaptéru mezi business komponentou a datovým zdrojem [32].

5.2.2.1 Doménové objekty

Doménové objekty jsou realizovány 14-ti perzistentními třídami v balíčku *cz.vutbr.fit.mis.dip.perfserver.model* serverové aplikace. Tyto třídy jsou anotovány pomocí JPA anotací. Finální schéma tříd je zobrazeno na obrázku 8.



Obrázek 8: Konceptuální diagram perzistentních tříd

5.2.2.2 Perzistence tříd

Perzistenci tříd zajišťuje framework Hibernate[17], který je definován v tzv. perzistenční jednotce souboru *persistence.xml* serverové aplikace (konkrétně je definován jako *org.hibernate.ejb.HibernatePersistence* v sekci <*provider*> této jednotky). Důležitou vlastností perzistenční jednotky je také vlastnost *hibernate.hbm2ddl.auto*, jejíž hodnota byla pro aplikaci nastavena na *update*. Tato hodnota zajistí vygenerování databázového schématu z perzistentních tříd

[17] implementován dle standardu JPA, http://www.hibernate.org/

při nasazaní serverové aplikace na aplikační server JBoss (děje se tak jen v případě, že toto schéma v prostředí PostgreSQL pro uvažovanou databázi ještě neexistuje). Diagram výsledného vygenerovaného databázového schématu z vytvořených perzistentních tříd je umístěn v přiloze C.

5.2.2.3 Realizace databázových dotazů

Pro databázové dotazy bylo upřednostněno JPQL nad Criteria API, kvůli jeho snadné čitelnosti dotazů. Databázové dotazy byly s jeho použitím definovány s využitím anotace *@NamedQuery*, kdy se u jejího atributu s názvem *name* uvádí název dotazu (lze se pak na něj odkazovat) a u autributu *query* pak samotný JPQL dotaz. Tyto dotazy se pak definují přímo u perzistetních tříd (Pokud je jich více, musí být definovány v rámci anotace *@NamedQueries.*), jak je vidět z ukázky na příkladu 14.

```
@Entity
@NamedQueries({
     @NamedQuery(name="projects", query="from Project"),
     @NamedQuery(name="projectsByName", query="from Project p where p.name =:
     project")
})
public class Project {
       // Definice třídy Project
}
```

Příklad 14: Ukázka definice JPQL dotazů s použitím anotací @NamedQuery a @NamedQueries

Kromě JPQL dotazů však bylo pro získání dat z databáze použito také jazyka SQL (JPQL totiž nedisponuje funkcí pro určení směrodatné odchylky, kterou bylo potřeba z dat určit). Dotaz s použitím toho jazyka se realizuje metodou s názvem *createNativeQuery* třídy *EntityManager*. Parametrem této metody je pak samotný dotaz v jazyce SQL.

5.2.2.4 DAO objekty

DAO objekty jsou implementovány v balíčku *cz.vutbr.fit.mis.dip.perfserver.dao.impl* tím způsobem, že každý tento objekt přistupuje k datům své databázové tabulky prostřednictvím implementovaného DAO rozhraní. Tato rozhraní jsou definovány v balíčku *cz.vutbr.fit.mis.dip.perfserver.dao*. Všechny DAO objekty jsou implementovány pomocí bezstavových bean.

Na příkladu 15 je vidět ukázka implementace DAO objektu *ProjectDaoImpl*, který implementuje rozhraní *ProjectDao*. Bezstavovost tohoto objektu je určena anotací *@Stateless* a anotace *@Inject* u jeho atributu *em* zajišťuje injekci instance entity manageru do tohoto atributu.

```
@Stateless
public class ProjectDaoImpl implements ProjectDao {
	@Inject
	private EntityManager em;
	// Následuje implementace metod rozhraní ProjectDao
}
```

Příklad 15: Ukázka implementace DAO objektu s názvem ProjectDaoImpl

5.2.3 Realizace business vrstvy a REST architektury

5.2.3.1 Business vrstva

Business vrstva byla z důvodu zamýšleného zpracování uživatelských akcí technologií AJAX realizována pomocí obyčejných managed bean se scope definovanou anotací *@ViewScoped*. Managed beana s tímto typem scope totiž umožňuje uchovávat stav managed beany mezi jednotlivými požadavky uživatele v rámci konkrétní webové stránky, což je pro správné fungovaní AJAXu ve spojení s PrimeFaces nezbytné. CDI managed beany nebyly použity z toho důvodu, že tento typ scope nepodporují.

Formuláře na webových stránkách jsou reprezentovány pomocí samostatných managed bean a omezení kladená na jejich jednotlivé prvky jsou specifikovány s využitím mechanismu Bean Validation popsaného ve specifikaci JSR 303.

DAO objekty jsou do managed bean injektovány pomocí anotace *@EJB* a případná managed beana reprezentující formulář pomocí anotace *@ManagedProperty* (v tomto případě je nezbytný také setter pro danou managed beanu). Ukázka managed beany realizující business logiku stránky *units.xhtml* i s injektováním zmíněných objektů je znázorněna na příkladu 16.

```
@ManagedBean
@ViewScoped
public class UnitBean implements Serializable {
     private static final long serialVersionUID = 1L;

     @ManagedProperty(value="#{unitForm}")
     private UnitForm form;
     @EJB
     private UnitDao unitDao;

     public void setForm(UnitForm form) {
            this.form = form;
     }
     // Následuje zbytek definice managed beany.
}
```

Příklad 16: Ukázka implementace managed beany s názvem unitBean

Ne všechny managed beany mají scope definovanou pomocí anotace *@ViewScoped*. Výjimkou je např. managed beana *ConfigBean*, která uchovává konfigurační nastavení pro celou aplikaci (scope má definován pomocí anotace *@ApplicationScoped)*. Managed beany, které se nepodílejí na AJAX funkcionalitě prezentační vrstvy jsou anotovány anotací *@RequestScoped*.

5.2.3.2 REST architektura

Pro realizaci komunikace serverové aplikace s klienty a s portálem qVue byla zvolena REST architektura dle specifikace JAX-RS. Ta je na JBoss AS realizována knihovnou RESTEasy[18]. Aby mohla být tato architektura, resp. její webové služby, v prostředí JBoss AS používány, je potřeba JAX-RS, resp. knihovnu RESTEasy, aktivovat třídou z příkladu 17.

```
import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/rest")
public class JaxRsActivator extends Application {}
```

Příklad 17: Ukázka třídy JaxRsActivator pro aktivaci knihovny RESTEasy

Třída JaxRsActivator z příkladu 17 musí pro splnění svého účelu dědit ze třídy Application. Řetězec /rest v její anotaci *@ApplicationPath* pak určí relativní adresu, na níž se namapuje JAX-RS servlet. Tímto způsobem tak bude daný řetězec vždy prefixem pro relativní URL adresu každé webové služby, která bude určena anotacemi *@Path*.

Realizace komunikace serverové aplikace s klienty

Komunikace serverové aplikace s klienty je realizována pomocí RESTful webových služeb implementovaných v balíčku *cz.vutbr.fit.mis.dip.perfserver.rest* ve třídě *RepRESTService*, která byla vytvořena jako singleton session beana.

Třída *RepRESTService* i s ukázkou implementace webové služby s názvem *processTestRunData*, která pomocí metody s názvem *processAttrResultData* zpracovává výkonnostní data zaslaná klientem, je ve zjednodušené formě zobrazena na příkladu 18.

[18] http://www.jboss.org/resteasy

```
@Path("/rep")
@Singleton
public class RepRESTService {
     // Definice atributů třídy a DAO objektů

     /*
       Metoda zpracovává výkonnostní data zaslaná od klienta, která jsou předána
       v parametru data. Pokud data nejsou v souladu s konfigurací serverové
       aplikace, metoda vrací instanci třídy ConfigProblem s popisem problému,
       jinak null.
     */
     private ConfigProblem processAttrResultData(TestRun testRun,
                                                 final AttrResultData data) {
            // Definice těla metody
     }

     @Path(RelURL.TEST_RUN + "{id}")
     @POST
     @Consumes("text/json")
     @Produces("text/json")
     public Response processTestRunData(
                                @PathParam("id") final Long testRunId,
                                final AttrResultData data) throws ParseException {

            TestRun testRun = testRunDao.getTestRunById(testRunId);
            ConfigProblem configProblem = processAttrResultData(testRun, data);
            if (configProblem == null) {
                   return Response.noContent()
                   .header("Message", "Data from PerfMonitor were stored.")
                   .status(201).build();
            } else {
                   return Response.ok(gson.toJson(configProblem))
                   .header("Message", "PerfMonitor is not correctly configured.")
                   .status(333).build();
            }
      }
      // Definice ostatních webových služeb
}
```

Příklad 18: Ukázka implementace webové služby processTestRunData ve třídě RepRESTService

U metody s názvem *processTestRunData* na příkladu 18 vidíme nezbytné JAX-RS anotace, které musely být použity proto, aby tato metoda byla schopna zpracovávat HTTP požadavky typu POST od klientů, kdy data v příchozích i odchozích HTTP zprávách jsou ve formátu JSON.

Z metody je rovněž patrné vytváření HTTP odpovědí klientům pomocí třídy *Response* a to včetně vytváření datového obsahu zprávy pomocí metody s názvem *ok* (), nového pole s názvem *Message* a HTTP status kódu.

Data třídy *ConfigProblem*, která metoda *processTestRunData* posíla klientovi zpět, jsou před samotným zasláním serializována do formátu JSON pomocí knihovny Gson.

Realizace exportu dat pro portál qVue

Export výkonnostních dat serverovou aplikací pro portál qVue implementuje metoda s názvem *getLastFinishedTestSuiteForPortal*, která je součástí třídy *RepRESTService*. Metoda využívá JAX-RS anotaci *@GET*. Ukázka formátu exportovaných dat při této operaci je dostupná na přiloženém DVD.

Proč byla zvolena REST architektura a jakým způsobem je řešena komunikace mezi klienty a implementovanými RESTful webovými službami serverové aplikace při testování platformy JBoss je dále vysvětleno v kapitole 6.

5.2.4 Realizace prezentační vrstvy

Prezentační vrstva byla implementována s použitím Facelets, JSF a PrimeFaces ve verzi 3.5, což byla nejnovější verze této knihovny v době implementace. Použití zmíněných technologií je stručně popsáno. V závěru této sekce jsou uvedeny ukázky náhledů na realizované srovnávací pohledy na výkonnostní data.

5.2.4.1 Webové stránky s použitím Facelets

Webové stránky byly postaveny na šabloně *default.xhtml*, která se nachází v adresáři *WEB-INF/templates* serverové aplikace. Tato šablona definuje záhlaví a tělo stránky (je určeno prvkem *div*, jehož atributu *id* má hodnotu *body*). Tělo se skládá z menu v levé části a z obsahové části vedle menu, která je definována značkou <*ui:insert*> z Facelets. Tato značka pak značí volitelnou část šablony, která je doplněna obsahem mezi značkami <*ui:define*> webové stránky (musí být shoda v hodnotě atributu name), která šablonu používá. To, že webová stránka používá šablonu, se specifikuje ve značce <*ui:composition*>.

Ilustrační ukázka ze šablony *default.xhtml*, která používá i komponenty z knihovny PrimeFaces (mají prefix p), zobrazující tělo této stránky je znázorněna na příkladu 19. Webová stránka, která zmíněnou šablonu *default.xhtml* používá je na příkladu 20.

```
<div id="body">
     <p:layout id="layout">
            <p:layoutUnit position="west">
                  <p:panelMenu>
                        <!-- Definice prvků menu -->
                  </p:panelMenu>
            </p:layoutUnit>
            <p:layoutUnit position="center">
                  <div id="content">
                        <ui:insert name="content">
                              <!-- Volitelná část šablony. -->
                        </ui:insert>
                  </div>
            </p:layoutUnit>
       </p:layout>
</div>
```

Příklad 19: Ukázka tzv. těla ze stránky default.xhtml

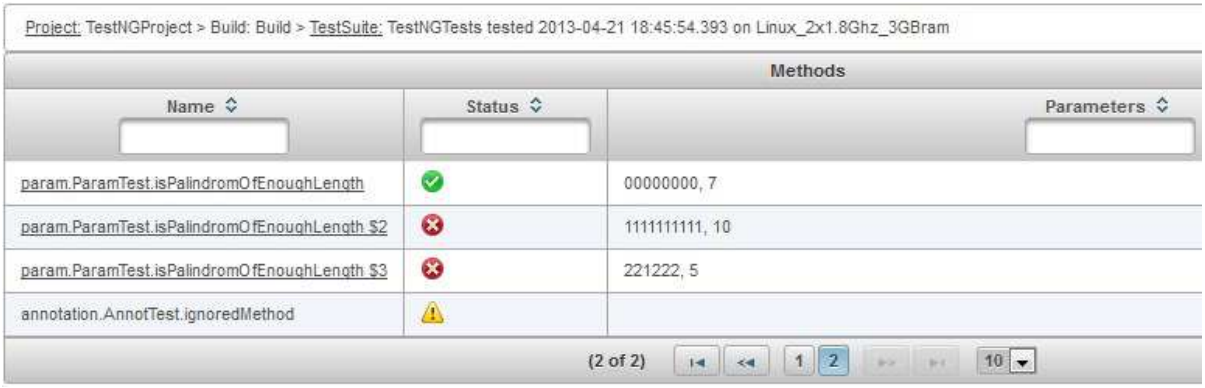
```
<?xml version="1.0" encoding="UTF-8"?>
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
     xmlns:ui="http://java.sun.com/jsf/facelets"
     xmlns:f="http://java.sun.com/jsf/core"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:c="http://java.sun.com/jsp/jstl/core"
     xmlns:p="http://primefaces.org/ui"
     template="/WEB-INF/templates/default.xhtml">

     <ui:define name="content">
            <!-- Definice obsahu stránky -->
     </ui:define>
</ui:composition>
```

Příklad 20: Webová stránka používající šablonu default.xhtml

5.2.4.2 Prezentace dat

Data na webových stránkách jsou prezentovány pomocí tabulek a grafů. Tabulky byly vytvořeny pomocí PrimeFaces datových tabulek užitím *<p:dataTable>*. Tyto tabulky umožňují také stránkování záznamů a jejich řazení a filtrování dle jednotlivých sloupců technologií AJAX, což bylo ve vhodných případech také použito. Ukázka realizované PrimeFaces tabulky se zmíněnými prvky ze stránky *methods.xhtml* je znázorněna na obrázku 9.



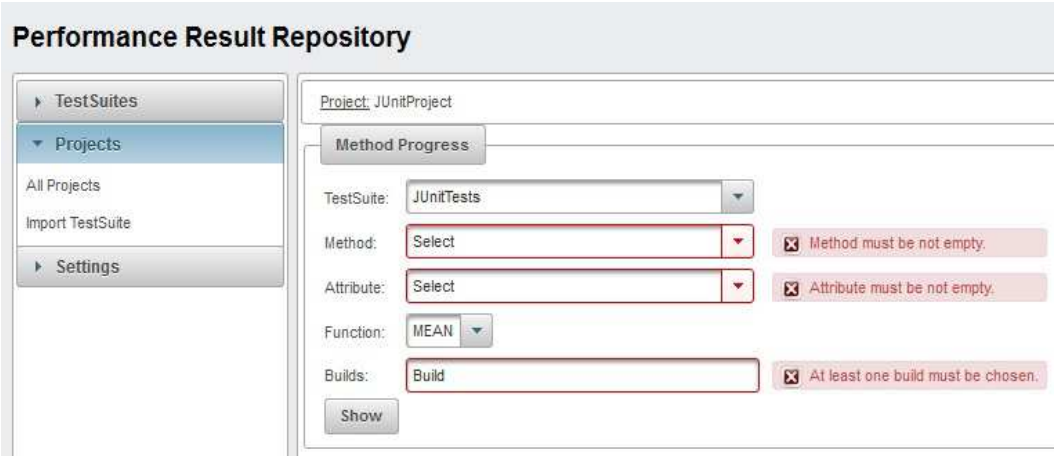
Obrázek 9: Náhled na tabulku s testovanými metodami ze stránky methods.xhtml serverové aplikace

V tabulce na obrázku 9 vidíme jednotlivé metody, které byly testovány v rámci zvolené test suity. Sloupec s označením *Status* informuje o výsledku metody (zelená značka značí úspěch, červená chybu a žlutá, že metoda byla při testování ignorována). Sloupec s názvem *Parameters* pak obsahuje výčet parametrů, se kterými byla konkrétní metoda volána.

Grafy v aplikaci byly implementovány také pomocí PrimeFaces knihovny. Pro Grafy ve srovnávacích pohledech typu I a III byla použita její značka *<p:barChart>* a pro graf na stránce *results.xhtml* značka *<p:lineChart>*. Grafy jsou realizovány jako interaktivní, což znamená, že u nich lze zvětšovat vybranou oblast a že v případě sloupcových grafů reagují na kliknutí myší.

5.2.4.3 Zpracování formulářů

Chybová hlášení při zpracování prvků formulářů jsou realizovány s využitím PrimeFaces komponenty *<p:message>*. Samotné zpracování formulářů je implementováno klasicky, výjimkou jsou jen formuláře na stránkách se srovnávacími pohledy na výkonnostní data, jejichž jednotlivé prvky jsou zpracovávány pomocí AJAXu. Ukázka kontroly formuláře s PrimeFaces chybovými hlášeními je na obrázku 10.



Obrázek 10: Náhled na formulář prošlý validací ze stránky view3.xhtml serverové aplikace

5.2.4.4 Import XML souboru test suity s výkonnostními daty

Import XML souboru test suity s výkonnostními daty je realizován na stránce *uploadfile.xhtml* pomocí PrimeFaces komponenty <*p:fileUpload*>. Tato komponenta ukládá uploadovaný souboru do atributu *file* třídy *UploadedFile* prostřednictvím setteru managed beany s názvem *fileUploadBean* (popsáno dle implementace). Před vlastním uložením uploadovaných dat do databáze je objekt třídy *UploadedFile* (atribut *file*) transformován na objekt třídy *File* a následně pomocí objektu třídy *SAXBuilder*[19] převeden na svou JDOM[20] reprezentaci a pársován.

Pokud nastane při pársování nějaka chyba, tato chyba je ve formě PrimeFaces hlášení zobrazena na webu a veškerá importovaná data jsou zahozena. V případě uspěšného pársování je celá test suita i se svými daty uložena v rámci svého projektu a buildu. Pro validaci importovaných XML souborů je použito XML schéma[21], které je definováno v adresáři *resources/files/schema.xsd* serverové aplikace. Příklad XML soubor test suity s výkonnostními daty je dostupný na přiloženém DVD.

[19] využívá SAX parser třetích stran, http://www.jdom.org/docs/apidocs/org/jdom2/input/SAXBuilder.html

[20] http://www.jdom.org

[21] http://www.w3schools.com/schema

6 Komunikační protokol

Tato část práce se zabývá volbou komunikační technologie, návrhem komunikačního protokolu mezi klientskou a serverovou aplikací a jeho vlastní implementací.

6.1 Analýza a návrh

6.1.1 Volba technologií pro komunikaci

Dle specifikace požadavků má klientská aplikace komunikovat se serverovou aplikací platformě nezávislou komunikací. Prostředků pro tento typ komunikace existuje celá řada (např. sockety, CORBA, XML-RPC, HTTP apod.). Vzhledem k tomu, že serverová aplikace má být také schopna poskytovat výkonnostní data portálu qVue (komunikuje pomocí HTTP protokolu), bude nejlepší na straně serverové aplikace pro obsluhu požadavků klienta zvolit webovou službu a s ní související komunikační protokol (tedy HTTP).

Webové služby mohou být realizovány dle dvou specifikací. Jsou to specifikace JAX-WS (JSR-224) a JAX-RS (JSR-331). Obě specifikace jsou koncepčně zcela odlišné a jsou vhodné pro webové služby, řešící odlišné problémy podle toho, zda webová služba pracuje spíše se zdroji či volá procedury. Obě specifikace se dále liší rozdílnou složitostí realizace klientů webové služby a také paměťovou a časovou režií, která je spojena s komunikací.

Na základě seznámení se s principy těchto specifikací se ukázalo jako vhodnější použít webovou službu dle specifikace JAX-RS (RESTful webová služba). Její přístup odpovídá práci se zdroji, což lépe postihuje řešený problém. Tato webová služba je také jednodušší z hlediska použití a její komunikace se vyznačuje menší datovou náročností, než u webové služby dle specifikace JAX-WS.

6.1.2 Princip řešení komunikace

Klient bude při testování odesílat získaná data prostřednictvím HTTP požadavků na URL adresy RESTful webových služeb serverové aplikace a ty s nimi budou provádět specifickou operaci dle jejich implementace (viz. podkapitola 5.2.3.2). Po provedení této operace bude vždy klientovi vrácena HTTP odpověď.

Přenášená data budou převážně komplexní, proto bude většina z těchto dat uložených v objektech. Samotný přenos dat bude realizován tak, že data v objektu budou ze strany klienta či serverovou aplikací serializována do textové podoby zvoleného formátu (JSON, XML apod.) a vložena do těla HTTP požadavku. Deserializace těchto dat do původního objektu je na straně webových služeb řešena implicitně. Klienti však tuto operaci budou muset provádět explicitně.

Vzhledem k možnému objemu přenášených dat bude data nejlépe odesílat prostřednictvím HTTP požadavku typu POST (umožňuje neomezený přenos dat). Pro přenos (export) dat určených pro portál qVue to bude zase s pomocí HTTP požadavku typu GET. Dá se totiž předpokládat, že tento export dat bude chtít uživatel také vyvolat s využitím webového prohlížeče.

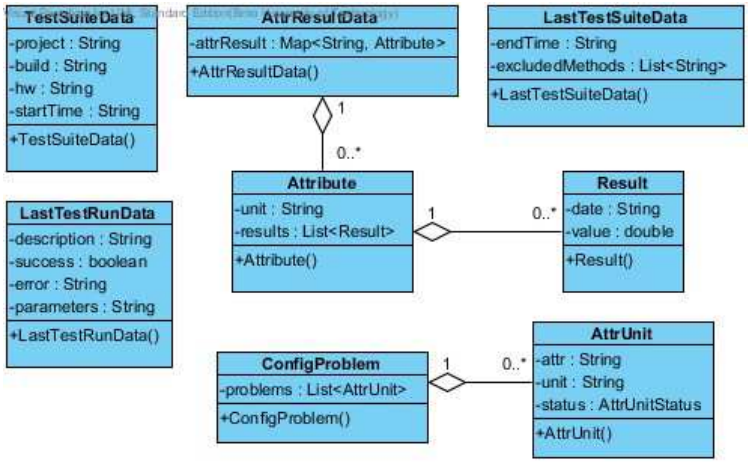
6.2 Implementace

V této části je popsána implementace komunikace všech entit (klienti, portál qVue) se serverovou aplikací. Pro přenos dat při komunikaci byl zvolen formát JSON.

6.2.1 Komunikace mezi klientem a serverovou aplikací

Tato část popisuje princip komunikace lokálních a vzdálených klientů se serverovou aplikací při testování platformy JBoss. Klienty se míní instance třídy *PerfMonitor*, jež je součástí projektu *PerfClient*, který byl vyvinut v rámci realizace klientské části aplikace.

Objekty, resp. třídy, které byly použity pro realizaci tohoto typu komunikace, jsou zobrazeny na obrázku 11.



Obrázek 11: Návrhový diagram tříd přenášených objektů mezi klientskou a serverovou aplikací

Význam jednotlivých tříd:

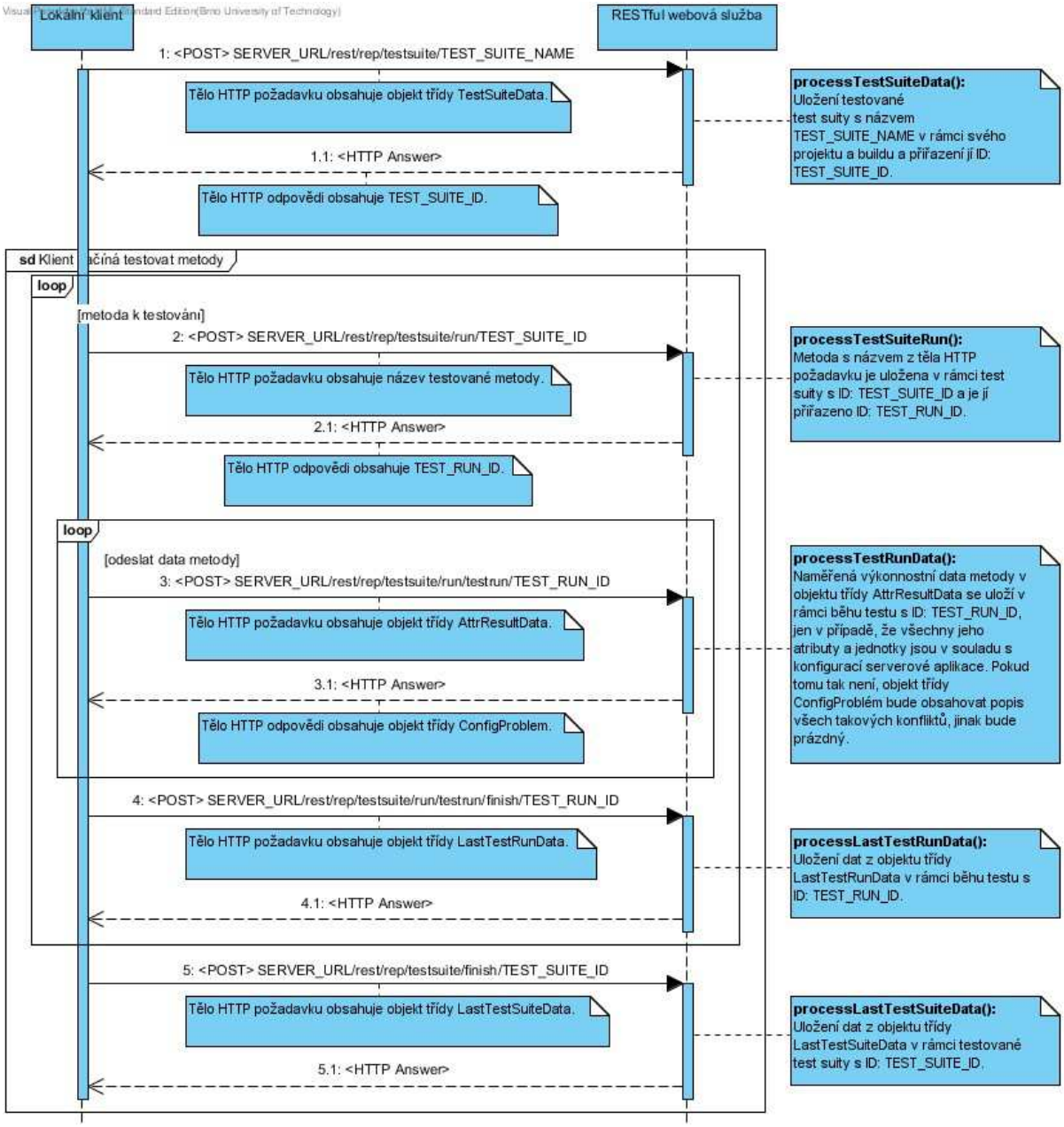
- **TestSuiteData** – Obsahuje základní data o testované test suitě (název projektu a buildu, použitou platformu a čas, kdy bylo testování test suity započato).
- **AttrResultData –** Třída slouží pro uložení naměřených výkonnostních dat. Představuje tedy buffer klienta.

- **ConfigProblem –** Tato třída slouží pro přenos možných konfiguračních problému klienta (pracuje s výkonnostním atributem či jednotkou nedefinovanou na serveru, či má atribut přiřazenou nesprávnou jednotku).
- **LastTestRunData –** Obsahuje údaje o testované metodě (popis metody, její výsledek, případnou výjimku a její parametry).
- **LastTestSuiteData –** Třída obsahuje čas, kdy byla test suita dotestována a případné metody, které byly vyloučeny z testování.

Komunikace lokálního klienta se serverovou aplikací

Tento typ komunikace je popsán sekvenčním diagramem na obrázku 12. Z diagramu je vidět, v jakém pořadí jsou posílány jednotlivé serializované objekty ve formátu JSON, prostřednictvím jakého HTTP požadavku a na jakou URL adresu. Vlastní obsluha HTTP požadavků je zajištěna webovými službami. Ty jsou popsány komentáři, kdy každý z těchto komentářů přibližuje název webové služby a jí provedenou akci. Obsah HTTP požadavků, resp. jejich datové části, je také popsán komentáři.

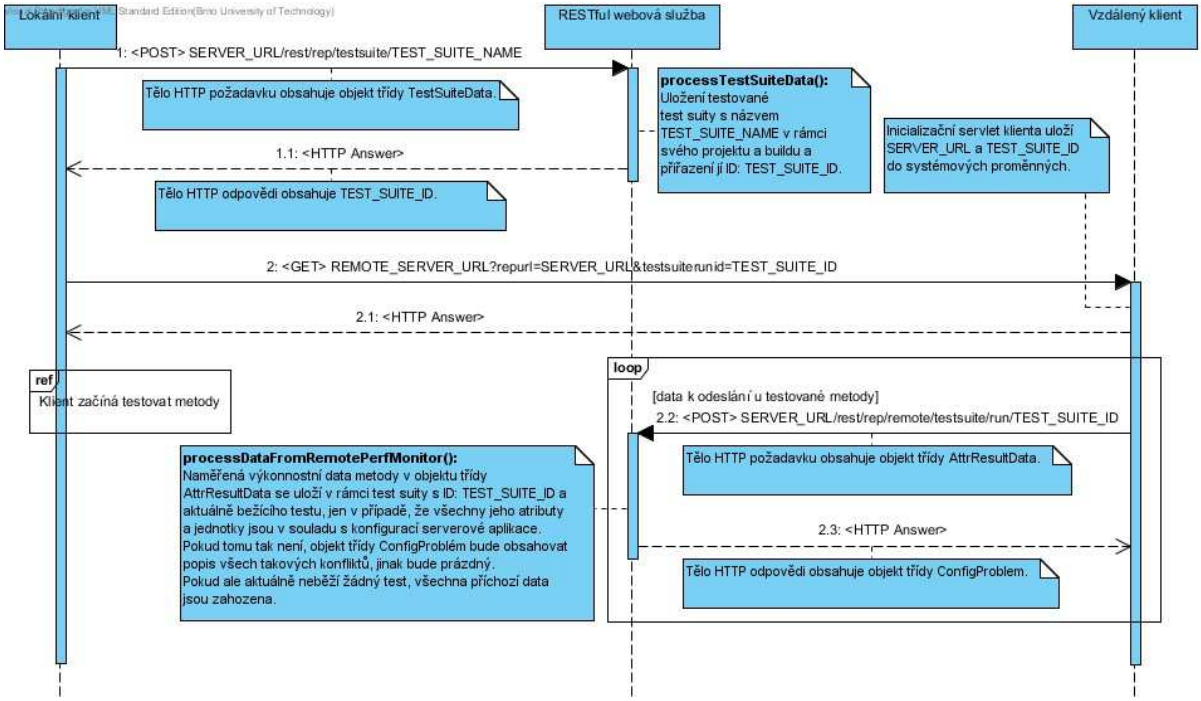
Při vlastní komunikaci se některá data posílají i v rámci uvažované URL adresy. To je patrné např. z prvního HTTP požadavku typu POST zaslaného klientem na URL adresu specifikovanou pomocí *SERVER_URL/rest/rep/testsuite/TEST_SUITE_NAME*. Při tomto požadavku klient předává serverové aplikaci ve svém těle serializovaný objekt třídy *TestSuiteData* a prostřednictvím URL název testované test suity (TEST_SUITE_NAME). Webová služba s názvem *processTestSuiteData*, pak s využitím DAO objektů tato data test suity uloží a v HTTP odpovědi klientovi vrátí ID, které bylo aktuálně testované test suitě přiřazeno (TEST_SUITE_ID). Dále pokračuje klient ve své činností tak, jak je zobrazeno v diagramu.



Obrázek 12: Sekvenční diagram komunikace lokálního klienta se serverovou aplikací

Komunikace vzdáleného klienta se serverovou aplikací

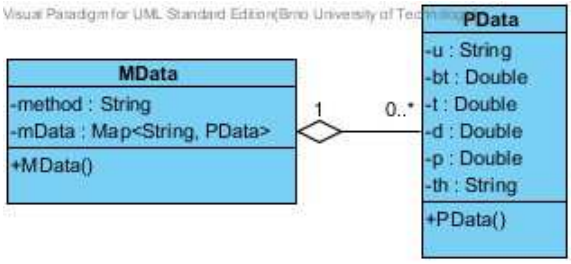
Na obráku 13 je znázorněna komunikace vzdáleného klienta se serverovou aplikací. V diagramu rovněž figuruje lokální klient, který nejdříve pro aktualně testovanou test suitu ve spolupráci se serverovou aplikací zjistí ID, které jí je v rámci testování přiřazeno (TEST_SUITE_ID), a poté toto ID zašle vzdálenému klientovi. Ten se poté může účastnit distribuovaných testů.



Obrázek 13: Sekvenční diagram komunikace vzdáleného klienta se serverovou aplikací

6.2.2 Komunikace portálu qVue se serverovou aplikací

Pro tento typ komunikace byly použity objekty, resp. třídy, znázorněné na obrázku 14.



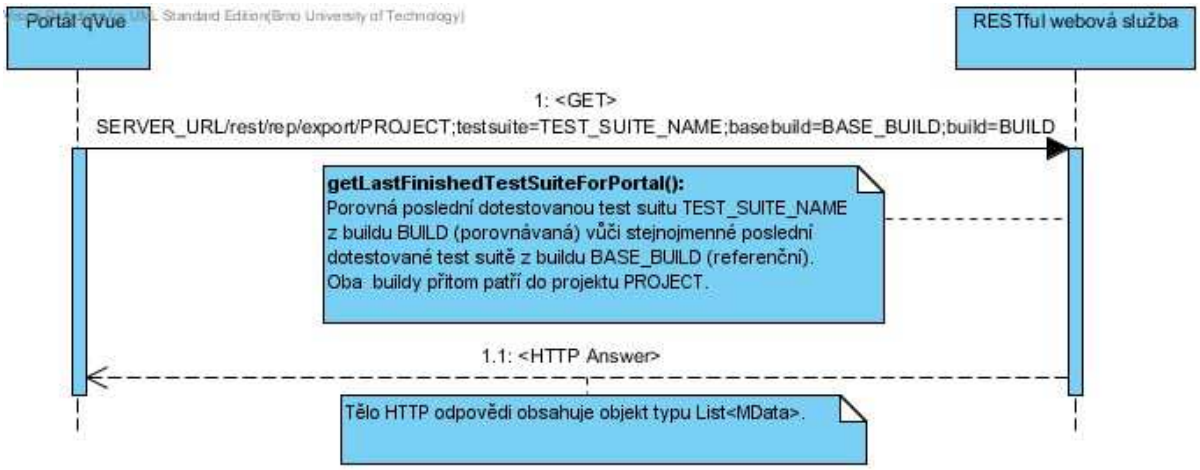
Obrázek 14: Návrhový diagram tříd přenášených objektů od serverové aplikace k portálu qVue

Význam tříd:

- **PData** – Obsahuje naměřené hodnoty určitého výkonnostního atributu z referenční test suity (bt) a srovnávané (t). Dále rozdíl těchto hodnot (d), relativní porovnání (p) a příznak možného překročení lokálního či globálního omezení (th).
- **MData** – Třída obsahuje srovnání určité metody z referenční a srovnávané test suity z hlediska všech výkonnostních atributů.

Vlastní komunikace portálu qVue se serverovou aplikací

Na obrázku 15 je znázorněn princip komunikace portálu qVue se serverovou aplikací. Portál qVue vysílá HTTP požadavek typu GET pro získání požadovaných dat (syntaxe tohoto požadavku je vysvětlena v příloze H - podkapitola 4.1) a serverová aplikace tato požadovaná data vrací v HTTP odpovědi, kdy tělo této odpovědi obsahuje serializovaný objekt typu List<MData>.



Obrázek 15: Sekvenční diagram komunikace portálu qVue a serverové aplikace

7 Testování projektu

Testování projektu si vzhledem k jeho složitosti vyžádalo komplexní přístup, kdy bylo nutno otestovat samotné API klienta, funkčnost klienta v JUnit a TestNG testech a také schopnost klienta a serverové aplikace účastnit se všech výkonnostních testů zmíněných v podkapitole 2.2.2. Dle zadání mělo být testování realizováno v testovacím prostředí platformy JBoss ve firmě Red Hat. Toto prostředí (včetně testů) však v době realizace práce ještě neexistovalo. Pro testování projektu bylo tedy použito vlastní testovací prostředí simulující tuto platformu. Příprava tohoto prostředí je popsána kapitole 7.1.

Samotné testování projektu pak proběhlo v realizovaném testovacím prostředí ve třech fázích s využitím vlastních výkonnostních JUnit a TestNG testů. Některé z těchto testů byly realizovány dle rozšířeného příkladu testování platformy JBoss uvedeného v podkapitole 2.2.3 této práce.

7.1 Příprava testovacího prostředí

Pro účely testování byla použita vlastní serverová aplikace *PerfServer* a její modifikovaná verze *TestedPerfServer*. Ta byla upravena tak, aby bylo možné u její stránky *results.xhtml* měřit response time a náročnost některých business operací pod ní. To si vyžádalo vložit do souboru *pom.xml* aplikace Maven závislosti pro moduly *PerfClientServlet* a *PerfRtFilter* a definovat *PerfRtFilter* pro stránku *results.xhtml* v souboru *web.xml*. Dále byl v souboru *ResultBean.java* aplikace *TestedPerfServer*, který realizuje business operace stránky *results.xhtml*, instanciován a volán klient tak, aby získal potřebná výkonnostní data[22]. PerfRtFilter měří výkonnostní atribut REMOTE_RESPONSE_TIME (viz. tabulka 5).

Aplikace *PerfServer* byla nasazena na OpenShift[23] a aplikace *TestedPerfserver* na JBoss AS na lokálním počítači. Před nasazením obou aplikací bylo nutno pro daný účel upravit jejich soubor *persistence.xml*. V případě lokálního nasazení této úpravě navíc předcházelo definování tzv. data source v aplikačním serveru.

Poté, co byly obě aplikace nasazeny na cílový server, v nich bylo potřeba definovat očekávané výkonnostní atributy spolu s jednotkami, které jim budou klienti zasílat. Pokud by totiž klient zasílal serverové aplikaci výkonnostní data s jí neznámými atributy či jednotkami, všechna tyto data by odfiltroval.

Atributy, které byly definovány v aplikaci *TestedPerfServer* jsou uvedeny v tabulce 4.

[22] jedná se o výkonnostní atributy uvedené v tabulce 5 kromě atributu REMOTE_RESPONSE_TIME

[23] dostupná na: http://app-cz0.rhcloud.com/PerfServer

Atribut	Jednotka atributu	Význam atributu
CPU_TIME	MILLISEC	Procesorový čas spotřebovaný testovanou metodou.
JVM_MEMORY_DELTA	KB	Změna využité JVM paměti před a po testu.
JVM_MEMORY	KB	Aktuální stav využité JVM paměti.
SYS_MEMORY	KB	Aktuální stav využité systémové paměti.
MANUAL_1	MILLISEC	Fiktivní manuální atribut.
MANUAL_2	KB	Fiktivní manuální atribut.

Tabulka 4: Atributy včetně jejich významu i jednotek definované v aplikaci TestedPerfServer

Atributy definované v aplikaci *PerfServer* jsou totožné s atributy u aplikace *TestedPerfServer* (s výjimkou atributů MANUAL_1 a MANUAL_2) a navíc jsou rozšířeny o atributy uvedené v tabulce 5.

Atribut	Jednotka atributu	Význam atributu
REMOTE_RESPONSE_TIME	MILLISEC	Response time změřený pomocí PerRtFilter.
REMOTE_JVM_MEMORY	KB	Aktuální stav využité JVM paměti zjištěný v metodě s anotací @PreDestroy.
REMOTE_TIME_FOR_PERF_DATA	MILLISEC	Čas načtení výkonnostních dat potřebných pro stránku results.xhtml z databáze.
REMOTE_JVM_MEMORY_USED_FOR_PERF_DATA	KB	Spotřebovaná JVM paměť pro výkonnostní data potřebná pro stránku results.xhtml.
REMOTE_TIME_FOR_STATS_OF_PERF_DATA	MILLISEC	Čas potřebný pro výpočet statistik z výkonnostních dat stránky results.xhtml.

Tabulka 5: Atributy včetně jejich významu i jednotek definované v aplikaci PerfServer

7.2 První fáze testování

Tato fáze testování proběhla pouze na lokálním počítači, kdy klient (aplikace *PerfClient*) odesílal výkonnostní data (všechny atributy z tabulky 4) z TestNG testů připraveného projektu *DataForComplexTest* do aplikace *TestedPerfServer*. Smyslem testů bylo ověřit funkčnost API klienta, pro získávání automatických a manuálních výkonnostních atributů, a schopnost klienta tyto data odeslat serverové aplikaci.

Testy zahrnovaly i takové případy, kdy klient během testu měřil automatické atributy (JVM_MEMORY a SYS_MEMORY) 10 krát, 100 krát a 1000 krát a takto zjištěná data pak téměř jednorázově odeslal. Tyto testy jsou označeny *autoAttributesN*, kde N je počet měření automatických atributů.

Všechna data, která klient během testování zaslal serverové aplikaci, jí byla přijata a tato fáze testování tedy proběhla úspěšně. V tabulce 6 jsou pro zajímavost uvedeny klientem naměřené výkonnostní charakteristiky u testů s označením *autoAttributesN*, které byly zjištěny z aplikace *TestedPerfServer*.

Metoda	**CPU_TIME** [10^{-3}s]	**JVM_MEMORY_DELTA** [KB]	**SYS_MEMORY** [KB] (změna)
autoAttributes10()	480	976	1612
autoAttributes100()	831	831	372
autoAttributes1000()	3220	3220	10540

Tabulka 6: Výkonnostní charakteristiky naměřené při testování u metod autoAttributesN()

U naměřených dat vidíme, že u testu s označením *autoAttributes10()* je hodnota atributu JVM_MEMORY_DELTA a změna SYS_MEMORY větší, než je tomu u testu *autoAttributes100()*, i když bychom očekávali vývoj opačný. Tento jev je zapříčiněn tzv. zahříváním systému, kdy při testování metody *autoAttributes10()* musela být prvotně inicializována paměť na lokálním a vzdáleném počítači, která má vždy větší režii, než při následujících inicializacích paměti.

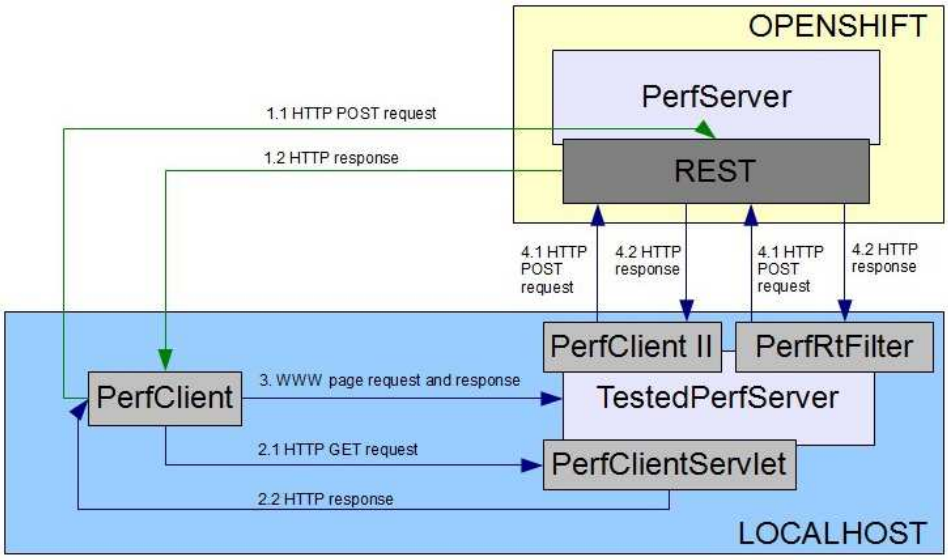
7.3 Druhá fáze testování

Ve druhé fázi testování byla použita serverová aplikace *TestedPerfServer* a nově i *PerfServer*. Aplikace *PerfServer* posloužila jako uložiště pro výkonnostní data, která jí zasílali klienti při nedistribuovaných a distribuovaných testech, a aplikace *TestedPerfServer* se stala předmětem distribuovaného testování.

7.3.1 Popis a princip testování

Testování v této fázi blíže popisuje obrázek 16. Na tomto obrázku jsou zachyceny všechny entity účastnící se testování včetně jejich vzájemné komunikace. Jsou zde aplikace *PerfServer* a *TestedPerfServer* (obsahuje komponenty *PerfClient II*, *PerfRtFilter* a *PerfClientServet)*, jež byly předpřipraveny v rámci přípravy testovacího prostředí (podkapitola 7.1) a *PerfClient* (klient v lokálním módu), který výkonává jednotkové testy (viz. dále).

Šipky na obrázku 16 ukazují směr komunikace mezi jednotlivými komponentami a popisky těchto šipek určují typ vyslaného HTTP požadavku, jestli se jedná o dotaz (request) nebo odpověď (response) a pořadí tohoto požadavku vůči ostatním požadavkům.



Obrázek 16: Schéma znázorňující průběh testování v prostředí simulujícím platformu JBoss

Realizované nedistribuované testy

Zelené šipky na obrázku 16 znázorňují komunikaci při testech, které nejsou distribuované. Při těchto testech probíhá komunikace pouze mezi klientem *PerfClient* a serverovou aplikací *PerfServer* (resp. její REST architekturou). PerfClient získává při těchto testech výkonnostní atributy uvedené v tabulce 4 (kromě atributů MANUAL_1 a MANUAL_2). Mezi tyto testy patří následující soubory z adresáře *src\test\java* projektů *JUnitTests* (obsahuje pouze JUnit testy) a *TestNGTests* (obsahuje pouze TestNG testy) :

- **annotation\AnnotTest.java** – Testy v tomto souboru testují podporu anotací, resp. atributů testu, pro ignorování testované metody, pro stanovení timeoutu a pro definování očekávané vyjímky u testované metody. V těchto testech není explicitně instanciován klient a měření výkonnostních atributu tedy probíhá pouze na začátku a na konci každého testu.
- **param\ParamTest.java** – Tato třída obsahuje parametrizovaný test. V JUnit jsou pro něj použity anotace *@RunWith* a *@Parameters*. V TestNG pak anotace *@DataProvider*.

- **simple.SimpleTest.java** – Obsahuje testy, které spadají do kategorie dlouhodobých testů, ale nejsou distribuované. Jeden z testů využívá rekurze, kdy při každém rekurzivním volání získá a odešle automaticky měřené výkonnostní atributy.

Realizované distribuované testy

U distribuovaných testů je komunikace reprezentována jak zelenými, tak modrými šipkami (viz. obrázek 16). Tyto testy se rovněž nacházejí v stejném adresáři projektů *JUnitTests* a *TestNGTests* jako předchozí testy a jsou specifikovány v souboru:

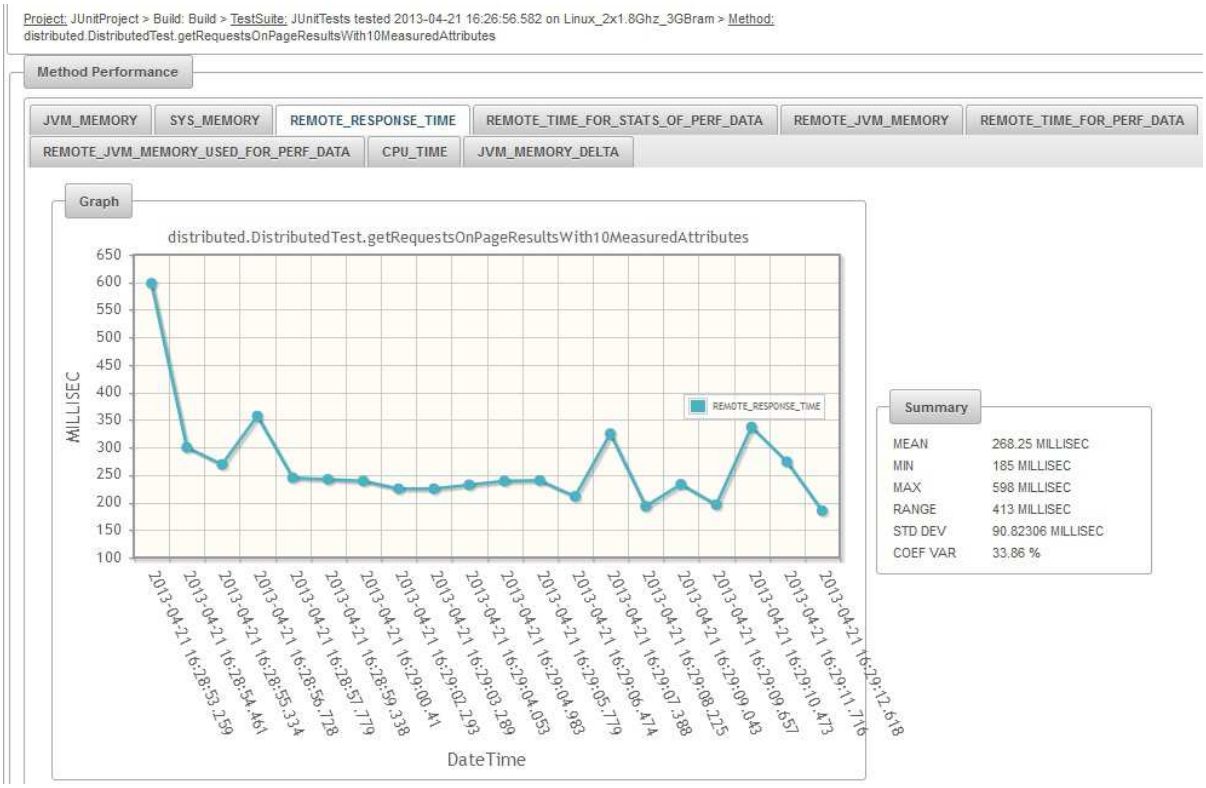
- **distributed\DistributedTest.java** – Třída obsahuje tři distribuované testy, kdy každý z nich vyšle 20 HTTP požadavků GET na stránku *results.xhtml* aplikace *TestedPerfServer*, jejíž data byla určena pomoci 10, 100 a 1000 měření automatických atributů v první fázi testování (Nazvěme tyto stránky podle počtu měření automatických atributů, tedy *results10.xhtml* apod. Metody, které testují tyto stránky *resultsN.xhtml* pak pojmenujme analogicky, tedy *20getReqOnResultsN*.). Během testování *PerfClient*, *PerfClient II* a *PerfRtFilter* odesílají průběžně dle schématu naměřená výkonnostní data. *PerfClient* odesílá atributy uvedené v tabulce 4 (kromě atributů MANUAL_1 a MANUAL_2). *PerfRtFilter* a *PerfClient II* odesílájí výkonnostní atributy, jež jsou popsány v podkapitole 7.1.

7.3.2 Výsledky distribuovaného testování

Zde jsou uvedena některá zajímavá výkonnostní data naměřená během distribuovaného testování projektu *JUnitTests*.

Response time u stránky results10.xhtml

Na obrázku 17 vidíme náhled z aplikace *PerfServer* na výsledky testu *20getRequestsOnResults10()*, který posílal 20 HTTP požadavků GET na stránku *results10.xhtml* aplikace *TestedPerfServer*. V grafu vidíme, jak se response time měnil v čase. Sekce Summary (vedle grafu) obsahuje základní statistické údaje, které byly z těchto dat zjištěny.

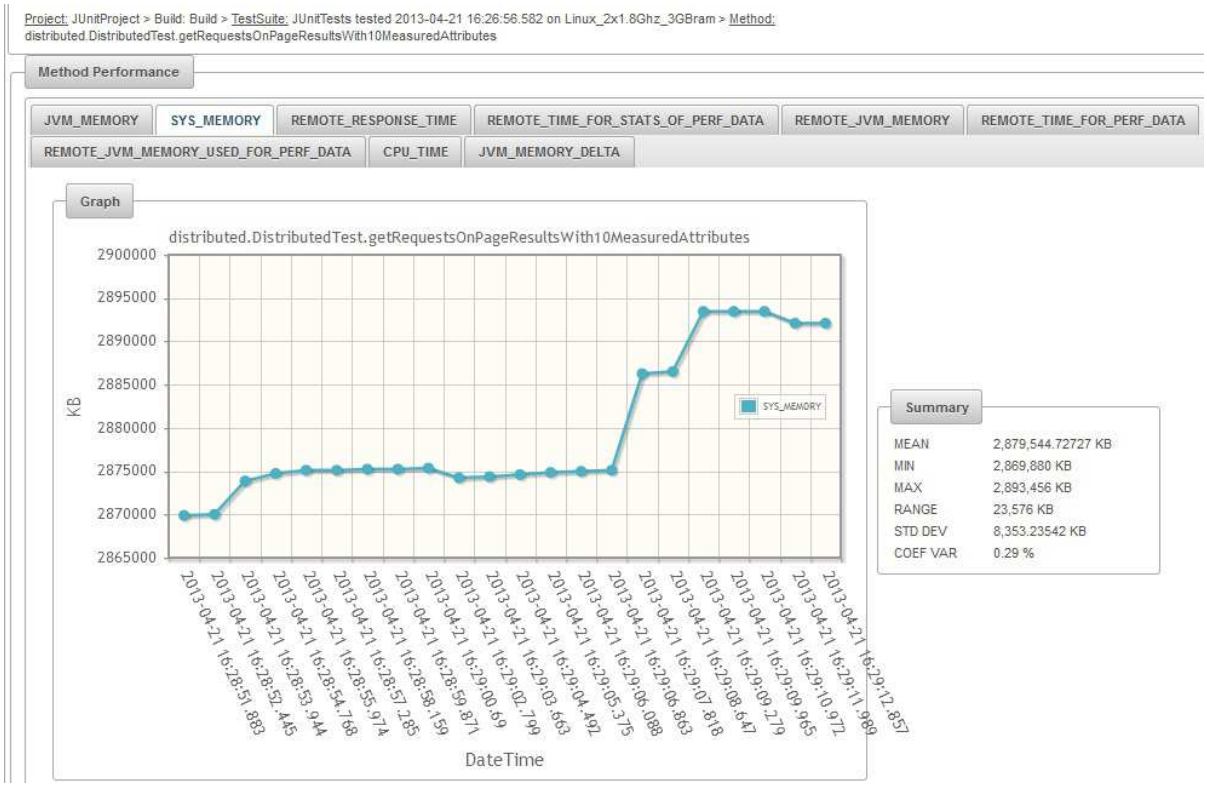


Obrázek 17: Náhled na vývoj atributu REMOTE_RESPONSE_TIME při testu 20getReqOnResults10()

Z vývoje atributu REMOTE_RESPONSE_TIME (response time) na obrázku 17 vidíme, že doba odezvy aplikačního serveru je pro první požadavek GET na stránku *results10.xhtml* největší a že se tato hodnota s každým nově zpracovaným požadavkem postupně ustaluje na nižší hodnotě.

Systémová paměť při testu 20getReqOnResults10()

Z náhledu aplikace *PerfServer* na obrázku 18 vidíme, jak se během uvažovaného testu vyvíjela systémová pamět na lokálním počítači.



Obrázek 18: Náhled na vývoj atributu SYS_MEMORY při testu 20getReqOnResults10()

Z vývoje atributu SYS_MEMORY (systémová paměť lokálního počítače) při testu *20getGetRequests()* na obrázku 18 vidíme, že do prvních 15 vyslaných GET požadavků klientem si systémová paměť udržuje téměř stabilní hodnotu a že při 16. požadavku již její hodnota prudce narůsta a pak se postupně ustaluje.

Jak je dále vidět z obrázku 18, serverová aplikace *PerfServer* nabízí stejné pohledy i na jiné změřené výkonnostní atributy. Nemá smysl je však všechny prezentovat.

Souhrnné výsledky distribuovaných testů

V tabulce 7[24] jsou uvedeny výsledky všech výkonnostních distribuovaných testů (jedná se o testy značené jako *20getReqOnResultsN()*), které byly získány v rámci testovaného projektu *JUnitTests*. U každého výkonnostního atributu je vždy uvedena pouze jeho průměrná hodnota bez jednotky (atribut je totiž s jednotkou svázán a jednotka je tak známa).

[24] Tabulka 7 se odvolává na tabulku 8, kde lze nalézt celé názvy výkonnostních atributů.

Atribut	20getReqOnResults10()	20getReqOnResults100()	20getReqOnResults1000()
A1	41 960	45 940	48 620
A2	-3 696	-18 087	79 835
A3	8 193,13636	44 145,90909	33 449,72727
A4	2 879 544,72727	2 919 358,36364	2 912 522,36364
A5	268,25	174	564,55
A6	154 915.73684	143 234	133 401,5
A7	70, 73684	57,3	159,8
A8	523,84211	1 587,55	6 593,65
A9	50,78947	39	43,3

Tabulka 7: Souhrnné výsledky distribuovaného testování metod 20getReqOnResultsN()

Celý název výkonnostního atributu	Atribut
CPU_TIME	A1
JVM_MEMORY_DELTA	A2
JVM_MEMORY	A3
SYS_MEMORY	A4
REMOTE_RESPONSE_TIME	A5
REMOTE_JVM_MEMORY	A6
REMOTE_TIME_FOR_PERF_DATA	A7
REMOTE_JVM_MEMORY_USED_FOR_PERF_DATA	A8
REMOTE_TIME_FOR_STATS_OF_PERF_DATA	A9

Tabulka 8: Celé názvy výkonnostních atributů s jejich zkratkami A1-A9

U naměřených dat je opět patrný jev spjatý se zahříváním systému, který lze nejvíce pozorovat v tabulce 7 u atributu REMOTE_RESPONSE_TIME u testovaných metod *20getReqOnResults10()* a *20getReqOnResults100()*.

7.4 Třetí fáze testování

Testování v této fázi proběhlo dle stejného schématu jako ve druhé fázi testování. Opět byly použity testy z projektů *JUnitTests* a *TestNGTests*. Účelem tohoto testování byla především příprava dat pro ověření aplikace *PerfServer* správně prezentovat výkonnostní data prostřednictvím srovnávacích pohledů. Pro tento účel byla výkonnostní data odesílaná během testování v rámci projektu s názvem *ViewProject*. Testy, resp. test suity, byly spouštěny s různým názvem v rámci různých buildů a

platforem. Veškeré toto testování proběhlo na operačním systému Ubuntu s výjimkou test suit s označením platformy Win7_2x1.8Ghz_3GBram, které byly skutečně testovány na operačním systému Windows 7. Tímto způsobem byla ověřena schopnost klienta pracovat ve dvou zcela odlišných operačních systémech a také to, že realizovaný způsob komunikace je nezávislý na použité platformě.

V rámci testování byly některé test suity spouštěny i souběžně, čímž se otestovala schopnost aplikace *PerfServer* přijímat data z paralelního testování více test suit. Pro některé test suity byl rovněž použit i uživatelem definovaný listener, který je v projektu *JUnitTests* definován v souboru */src/main/java/listener/UserJUnitPerfListener.java* (v projektu *TestNGTests* tento listener nese analogický název). Díky tomuto listeneru a standardnímu listeneru implicitně definovaném v klientovi se mohly v rámci různých běhů získávat stejné test suity odlišné množiny výkonnostních atributů, což umožnilo následně v aplikaci *PerfServer* testovat srovnávací pohledy na výkonnostní data i v situacích, kdy u jedné test suity byl atribut měřen a u druhé ne.

Ukázky některých možných pohledů na výkonnostní data z této fáze testování jsou uvedeny v příloze D, E a F.

8 Závěr

Cílem této diplomové práce bylo vyvinout aplikaci typu klient-server pro sběr a srovnávání výkonnostních dat z testovacího procesu platformy JBoss ve firmě Red Hat. Úkolem klientské aplikace bylo při testování platformy JBoss získávat uživatelem definovaná výkonnostní data a průběžně je odesílat serverové aplikaci platformě nezávislou komunikací. Serverová aplikace postavená na technologii Java EE měla sloužit jako uložiště těchto dat a prostřednictvím webového klienta podporovat několik náhledů na daná data. Tyto náhledy umožní uživateli snadno porovnávat výkon testovaných metod v různých buildech, resp. jejich test suit, z hlediska zvolených výkonnostních atributů a testovacích platforem (testovací platforma je určena hardwarem počítače a jeho operačním systémem), na nichž testování probíhalo. Tímto způsobem serverová aplikace umožní identifikovat výkonnostní problémy v testovaných aplikacích a nepřímo pomůže v jejich odstranění. Díky daným vlastnostem se aplikace stane cenným nástrojem pro vytváření výkonnostně optimalizovaného softwaru.

Během vývoje byly do aplikace zapracovány téměř všechny formální i technické požadavky stanovené zadavatelem. Mimo tyto požadavky byly do aplikace implementovány také některá vlastní vylepšení (Grafy umožňující po kliknutí na hodnotu atributu přejít na jeho detailní stránku. Informování klientů serverovou aplikací o jejich případně nesprávné konfiguraci apod.).

Vývoj aplikace probíhal tím způsobem, že byl zadavatel průběžně seznamován s její klientskou i serverovou částí. Serverová aplikace byla vždy nasazena na platformu OpenShift a pomocí klienta a jeho komponent naplněna ukázkovými daty. Zadavatel si ji poté mohl vyzkoušet a konfrontovat své představy o aplikaci s jejím aktuálním stavem. Serverová aplikace byla spolu s klientem a jeho komponentami úspěšně otestována v prostředí simulujícím platformu JBoss, čímž byla prokázána technická způsobilost celé aplikace pro reálné použití v praxi.

V dokončovací fázi této práce byl projekt představen také pracovním kolegům zadavatele, kteří navrhli vhodná rozšíření pro serverovou aplikaci. Ne všechna z těchto rozšíření se vzhledem k jejich složitosti a dosavadnímu rozsahu práce podařilo zapracovat. Rozšíření, která nebyla implementována, se týkají srovnávacích pohledů na výkonnostní data. Tyto pohledy by měly umožňovat určovat průměrnou hodnotu výkonnostních atributů u každé metody ne pouze z jednoho konkrétního běhu test suity, ale i z více (vždy v rámci uvažované testovací platformy). Tímto způsobem se vyřeší problém se zahříváním systému (zmíněn v podkapitolách 7.2 a 7.3.2), který výrazně ovlivňuje naměřená výkonnostní data a tím i relevantnost výkonnostního srovnávání, ale eliminují se také rozdíly v naměřených hodnotách, které mohli vzniknout v důsledku stavu operačního systému a jeho přístupu k řízení zdrojů v čase testování.

Další navržené rozšíření se týká konkrétně srovnávacího pohledu typu II, který by měl nově umožňit výkonnostní srovnání více zvolených běhů test suit s referenčním během test suity a to nejen z pohledu jednotlivých metod, ale z pohledu test suit jako celků.

Vhodné by bylo realizovat také nový typ pohledu, který by umožňoval výkonnostní atributy naměřené v určitém testu libovolně seskupovat a zobrazovat v rámci jednoho grafu, případně pouze v tabulce. Takový pohled by byl užitečný v případě, kdybychom v testu měřili např. response time u webové stránky, pro jejíž vygenerování by bylo nutné získat data z databáze a následně je transformovat. Pokud bychom tyto operace změřili a vybrali si pro náš pohled čas databázového dotazu a čas transformace, snadno bychom s užitím tohoto pohledu určili, který z naměřených časů response time ovlivňuje nejvíce apod.

Posledním navrženým rozšířením je možnost definování hraničních výkonnostních omezení pro jednotlivé test suity, které se u srovnávacího pohledu typu II budou aplikovat analogicky jako např. globální hraniční výkonnostní omezení.

Zdrojové kódy finální verze vyvinuté aplikace jsou dle požadavku zadavatele zveřejněny na serveru GITHUB (`https://github.com/xjara/perf-result-repository`).

Literatura

[1] Unit testing. In: *Wikipedia* [online]. 2013-05-01 [cit. 2013-05-14]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Unit_testing>.

[2] Programování řízené testy. In: *Wikipedia* [online]. 2013-03-21 [cit. 2013-05-14]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Programování_řízené_testy>.

[3] MAŘÍK, Radek. *Testování jednotek* [online]. 2007 [cit. 2013-05-14]. Dostupné z WWW: <http://labe.felk.cvut.cz/~marikr/teaching/Y33TSW_10/11.testovaniJednotek.pdf>.

[4] JUnit. In: *Wikipedia* [online]. 2013-04-22 [cit. 2013-05-14]. Dostupné z WWW: <http://en.wikipedia.org/wiki/JUnit>.

[5] SUnit. *Camp Smalltalk* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://sunit.sourceforge.net/index.html>.

[6] JUnit wiki. In: *GitHub* [online]. [cit. 2013-05-14]. Dostupné z WWW: <https://github.com/junit-team/junit/wiki>.

[7] RunListener. *JUnit API* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://junit.sourceforge.net/javadoc/org/junit/runner/notification/RunListener.html>.

[8] BEUST, Cédric. *TestNG* [online]. 2004, 2013-03-30 [cit. 2013-05-14]. Dostupné z WWW: <http://www.testng.org>.

[9] ITestListener. *TestNG API* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://testng.org/javadocs/org/testng/ITestListener.html>.

[10] MOLYNEAUX, Ian. *The Art of Application Performance Testing*. Sebastopol: O'Reilly Media, 2009, 133 s. ISBN 978-0-596-52066-3.

[11] SIGAR API. *Hyperic* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://www.hyperic.com/products/sigar>.

[12] Welcome to Apache Maven. *Apache Maven Project* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://maven.apache.org>.

[13] SRIRAGAN. *Apache Maven 3 Cookbook*. Birmingham B3 2PB, UK: Packt Publishing, 2011. ISBN 978-1-849512-44-2.

[14] Maven Surefire Plugin. *Apache Maven Project* [online]. 2013-05-05 [cit. 2013-05-14]. Dostupné z WWW: <http://maven.apache.org/surefire/maven-surefire-plugin>.

[15] Gson User Guide. *Gson* [online]. [cit. 2013-05-14]. Dostupné z WWW: <https://sites.google.com/site/gson/gson-user-guide>.

[16] Response Time Filter. JBOSS. *RHQ* [online]. 2008-03-29 [cit. 2013-05-14]. Dostupné z WWW: <https://docs.jboss.org/author/display/RHQ/Response+Time+Filter>.

[17] SLF4J. *SLF4J Project* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://www.slf4j.org>.

[18] Apache log4j 1.2. *Apache logging services* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://logging.apache.org/log4j/1.2>.

[19] Jendrock E., Evans I. et al. *The Java EE 6 Tutorial: Basic Concepts*. ISBN-13: 978-0137081851

[20] Servlet. *Servlet API documentation* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://tomcat.apache.org/tomcat-5.5-doc/servletapi/javax/servlet/Servlet.html>.

[21] Why PrimeFaces. *PrimeFaces* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://www.primefaces.org/whyprimefaces.html>.

[22] PRIME, Optimus. *PrimeFaces User's Guide 3.5* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://primefaces.googlecode.com/files/indexed_primefaces_users_guide_3_5.pdf>.

[23] Representational state transfer. In: *Wikipedia* [online]. 2013-05-13 [cit. 2013-05-14]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Representational_state_transfer>.

[24] Jersey 1.17 User Guide. *Jersey* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://jersey.java.net/nonav/documentation/latest/index.html>.

[25] Java EE 6 APIs. *The Java EE 6 tutorial* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://docs.oracle.com/javaee/6/tutorial/doc/bnacj.html>.

[26] Java EE. In: *Wikipedia* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/Java_EE>.

[27] Getting Started with JBoss Application Server 7. JBOSS. *JBoss AS 7.1 Documentation* [online]. 2012-12-14 [cit. 2013-05-14]. Dostupné z WWW: <https://docs.jboss.org/author/display/AS71/Getting+Started+Guide>.

[28] JBoss AS7 Deployment Plugin. JBOSS. JBoss Application Server 7 Maven Plugin [online]. 2013-02-25 [cit. 2013-05-14]. Dostupné z WWW: <https://docs.jboss.org/jbossas/7/plugins/maven/latest/index.html>.

[29] PostgreSQL. In: *Wikipedia* [online]. 2013-05-09 [cit. 2013-05-14]. Dostupné z WWW: <http://cs.wikipedia.org/wiki/PostgreSQL>.

[30] JBoss Tools. JBOSS. *JBoss Community* [online]. [cit. 2013-05-14]. Dostupné z WWW: <http://www.jboss.org/tools>.

[31] Get Started with Openshift. RED HAT. *OpenShift* [online]. [cit. 2013-05-14]. Dostupné z WWW: <https://www.openshift.com/get-started>.

[32] Alur D., Crupi J. et al. *Core J2EE patterns: best practices and design strategies*. ISBN 01-314-2246-4.

Seznam příloh

Příloha A – Specifikace požadavků
Příloha B – Diagram tříd klientské aplikace
Příloha C – Databázové schéma serverové aplikace
Příloha D – Ukázka náhledu na stránku view1.xhtml (pohled typu I)
Příloha E – Ukázka náhledu na stránku view2.xhtml (pohled typu II)
Příloha F – Ukázka náhledu na stránku view3.xhtml (pohled typu III)
Příloha G – Manuál pro klientskou aplikaci
Příloha H – Manuál pro serverovou aplikaci
Příloha I – Obsah přiloženého DVD
Příloha J – DVD

Příloha A - Specifikace požadavků

Zde jsou detailně zaznamenány požadavky na vyvíjenou aplikaci typu klient-server. Tyto požadavky byly konzultovány na několika sezeních se zadavatelem před samotným zahájením projektu, ale také během jeho vývoje.

Požadavky na klientskou část aplikace

- Klientská aplikace bude implementována v jazyce Java.
- Aplikace musí být schopna měřit uživatelem definované výkonnostní atributy (JVM paměť, systémovou paměť, procesorový čas apod.) v rámci JUnit a TestNG testů. Tyto testy budou součástí projektu, který bude využívat nástroj Maven.
- Projekt, jeho verze (tzv. build), test suita a užitá testovací platforma, v rámci nichž budou testy spuštěny, budou klientovi předány před zahájením testování. Pokud některé z těchto specifik nebude zadáno, bude pro testování použit standardní listener neprovádějící výkonnostní testování.
- Každý výkonnostní atributu pak bude svázán s jednotkou, v níž bude měřen.
- Klient bude standardně měřit procesorový čas a využitou JVM paměť v každém jednotkovém testu a to bez toho, aby byl v testech explicitně instanciován a volán.
- Výkonnostní atributy bude moci uživatel definovat jako automaticky měřené či manuálně. Automatickými atributy se rozumí ty, které bude přímo zjišťovat a ukládat do své paměti klient. Implicitně budou měřeny vždy na začátku a konci každé testované metody. Při každém takovém měření se vždy určí hodnoty všech definovaných automatických atributů. Manuální atributy jsou pak takové, jejichž hodnota bude zjištěna z vnějšku klienta (např. čas odezvy webového serveru) a prostřednictvím jeho metody pouze vložena do jeho paměti.
- U každého výkonnostního atributu je požadováno určit čas, kdy byla jeho hodnota získána.
- Oba typy atributů budou moci být měřeny a ukládány do paměti klienta kdekoliv uvnitř každého testu. Obsah paměti klienta bude moci být na požádání odeslán serverové aplikaci uvnitř každého testu, tak aby i při dlouhodobých testech byly v serverové aplikaci k dispozici co nejaktuálnější data o průběhu testování.
- API klienta, které bude zajišťovat měření výkonnostních data, jejich ukládání do bufferu a jeho následné odeslání, by mělo být pro uživatele flexibilní z hlediska použití a nemělo by jej při měření výkonnostních dat nijak omezovat.

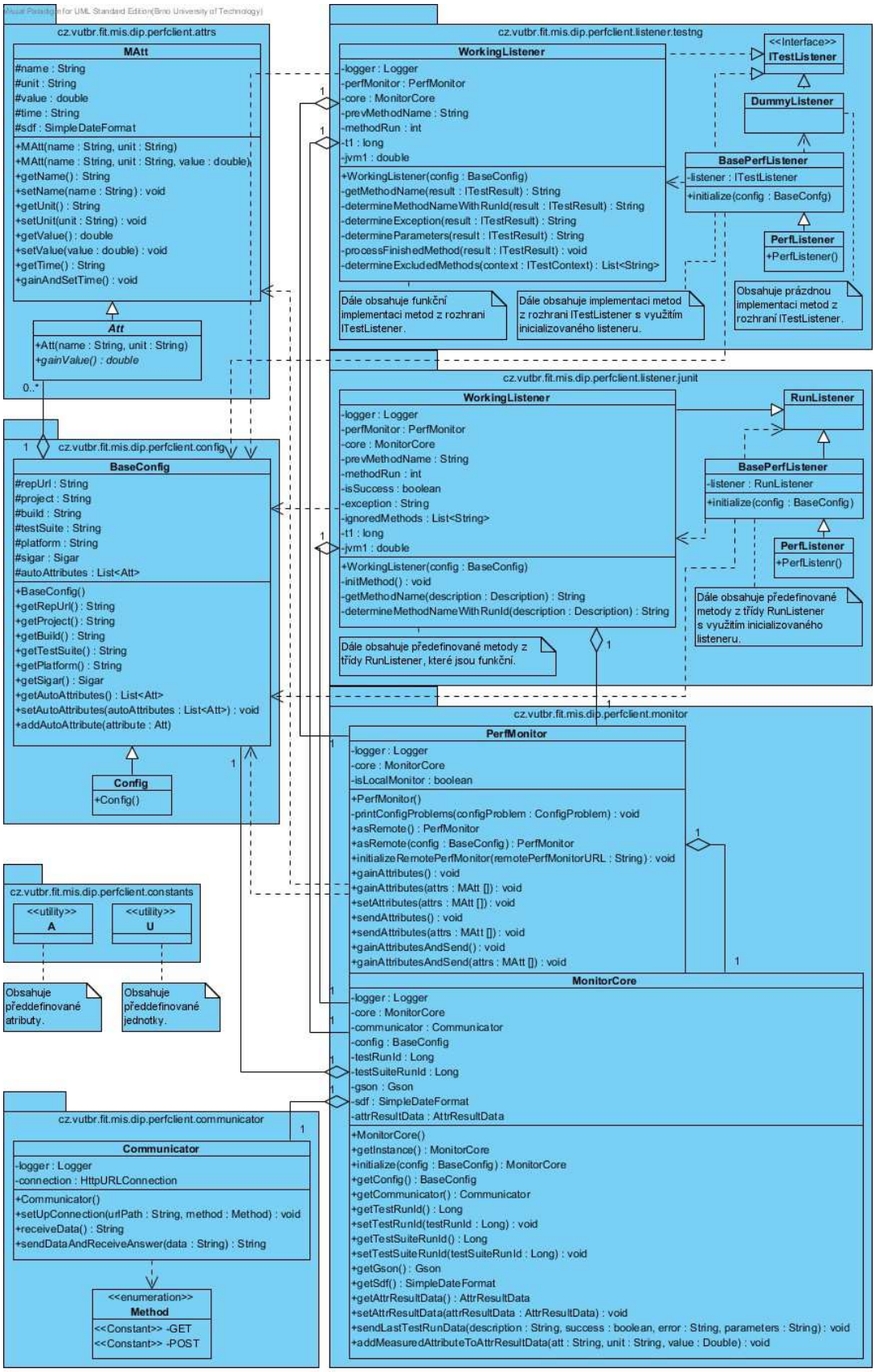
- Kromě výkonnostních atributů by klient měl také sbírat informace o tom, jestli testované metody dopadly úspěšně či nikoliv. V případě neúspěchu by měla být odeslána také výjimka, při níž chyba vznikla.
- Klient by měl také určit časové období, ve kterém byla test suita testována.
- Komunikace mezi klientem a serverovou aplikací bude realizována tak, aby byla nezávislá na platformě. Průběh komunikace bude ze strany klienta logován.
- V případě, že bude klient nakonfigurován jiným způsobem než serverová aplikace (myslí se tím atributy a jednotky, v nichž se atributy měří), klient tento problém oznámí uživateli a to včetně detailního popisu chyb, tak aby uživatel věděl, jak konfiguraci klienta napravit, aby byla v souladu s konfigurací serverové aplikace.
- Je vyžadováno, aby se jednoho testovacího případu mohlo účastnit současně více klientů (případ distribuovaného testování), kdy každý z nich bude pracovat na jiném fyzickém počítači a bude měřit a odesílat své specifické výkonnostní atributy (automatické i manuální).
- S využitím klienta by měl být také vytvořen nástroj, který umožní měřit dobu odezvy pro požadované stránky testované webové aplikace na webovém serveru (tzv. response time) v rámci testu, který je spuštěn na jiném počítači. Tento nástroj by měl umožnit snadnou integraci do testované webové aplikace.

Požadavky na serverovou aplikaci

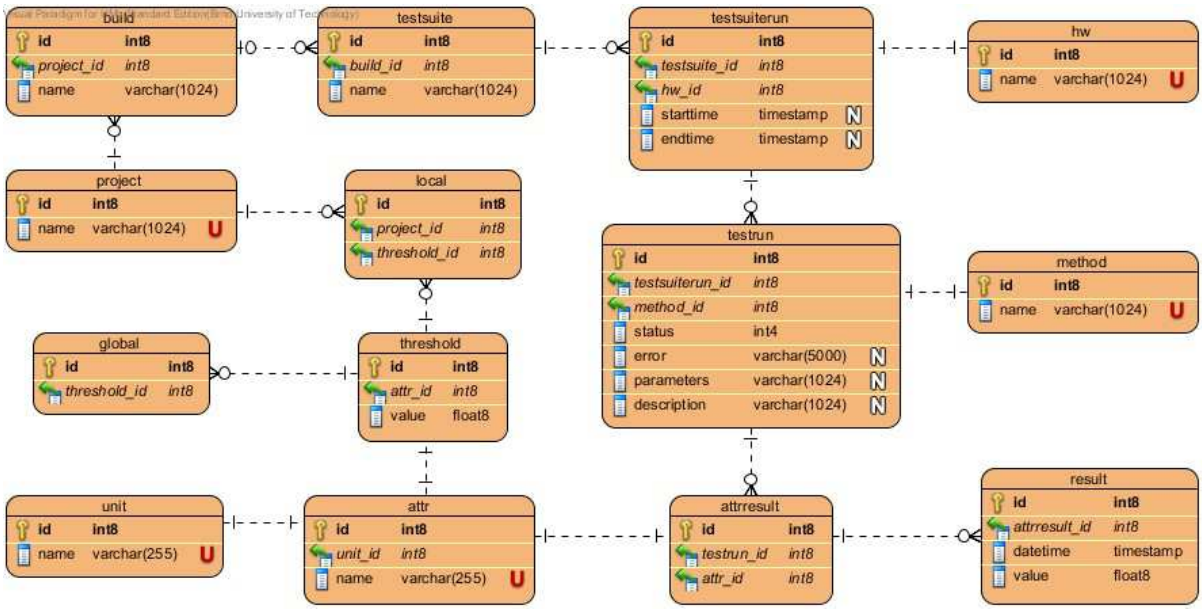
- Serverová aplikace bude implementována s využitím technologie Java EE.
- Aplikace musí být schopna uchovávat výsledky výkonnostního testování pro různé projekty. Každý projekt přitom během svého vývoje prochází evolučními změnami, které jsou zachyceny v jeho jednotlivých buildech. Buildy se skládají z balíčků obsahujících třídy.
- Testy jsou spouštěny ve formě test suit, které obsahují testované třídy, resp. jejich metody. Každá test suita přitom může být testována opakovaně. Je tedy žádoucí, aby aplikace poskytovala uživateli informaci v jakém časovém rozmezí se ta či ona test suita testovala. Případně, jestli se test suita ještě testuje.
- Aplikace by měla umožnit uživateli snadnou orientaci v právě testovaných test suitách či dokončených dle zvoleného časového období.
- Test suity jsou testovány na různých platformách. Ty jsou identifikovány svým hardwarem a instalovaným operačním systémem.
- Je vyžadováno, aby aplikace podporovala jednorázové, dlouhodobé a distribuované testy.
- Testování metod muže trvat několik hodin. Bude tedy vhodné u metod zobrazovat stav testování (tedy jestli se právě testuje, či již byla dotestována a s jakým výsledkem). Je třeba počít také s možností, že metoda bude při testování ignorována.

- Uživatel bude moci v serverové aplikaci definovat výkonnostní atributy a jejich jednotky, které se budou při jakémkoliv testování uvažovat. Výskyt těchto atributů a jejich jednotek se bude kontrolovat v příchozích datech od klientů. V Případě, že se v těchto datech vyskytne nějaký atribut nebo jeho jednotka, které nejsou serverovou aplikací podporovány, všechna tato příchozí data budou serverem ignorovány a klient bude následně na danou skutečnost upozorněn detailní popisem chyby. Tento popis chyby by měl uživateli klienta pomoci v napravení jeho konfigurace, tak aby byla v souladu s nastavením serverové aplikace.
- Serverová aplikace by měla na každou klientovu zprávu reagovat návratovým kódem, tak aby uživatel klienta věděl, jestli serverová aplikace zprávu úspěšně zpracovala či ne.
- U každého projektu je požadována možnost nastavit hraniční hodnotu pro každý definovaný výkonnostní atribut (lokální omezení). Tyto hraniční hodnoty budou sloužit jako indikátory značící případné překročení max. tolerovatelného rozdílu hodnot atributů u porovnávaných běhů test suit.
- Hraniční hodnoty pro výkonnostní atributy budou moci být nastaveny také globálně pro všechny projekty (globální omezení). Pokud budou pro atribut nastaveny oba typy omezení, vyšší prioritu bude mít lokální (předpokládá se, že má hraniční hodnoty nastaveny přísněji).
- Serverová aplikace musí být schopna na požádání odeslat srovnání dvou naposledy dotestovaných test suit z hlediska všech výkonnostních atributů ze dvou zadaných buildů v rámci určitého projektu. Srovnání bude zahrnovat hodnoty atributů z obou test suit, jejich rozdíl, relativní porovnání i případný příznak překročení globálního či lokálního omezení atributu. Tato funkcionalita bude sloužit pro integraci s portálem qVue.
- Prostřednictvím webového klienta musí serverová aplikace uživateli umožnit tři typy srovnávacích pohledů na naměřená výkonnostní data. Pohledy budou prezentovány prostřednictvím grafů a tabulek.
- Kromě srovnávacích pohledů by aplikace měla umožnit také pohled na výkonnostní data u zvolené metody z hlediska měřených atributů. Data by měla být prezentována formou grafu a tabulky. Pro data by měly být také určeny základní statistické údaje. Tento pohled bude ještě zahrnovat případný popis metody a výjimku (pokud metoda skončila chybou).
- Aplikace musí také umožnit manuální import výkonnostních dat test suity v podobě XML souboru. Zde se očekává, že aplikace bude zpracovávat výkonnostní atributy a jejich jednotky v soboru stejným způsobem jako při přijímání dat od klienta (pokud bude rozpor v konfiguraci, žádná data se neuloží). Tato funkcionalitu uživatel využije v případě, že výkonnostní data u test suity získal jiným způsobem než pomocí vyvíjeného klienta.
- Poslední požadavkem u aplikace je, aby umožňovala mazat celou test suitu či celý projekt včetně všech souvisejících dat.

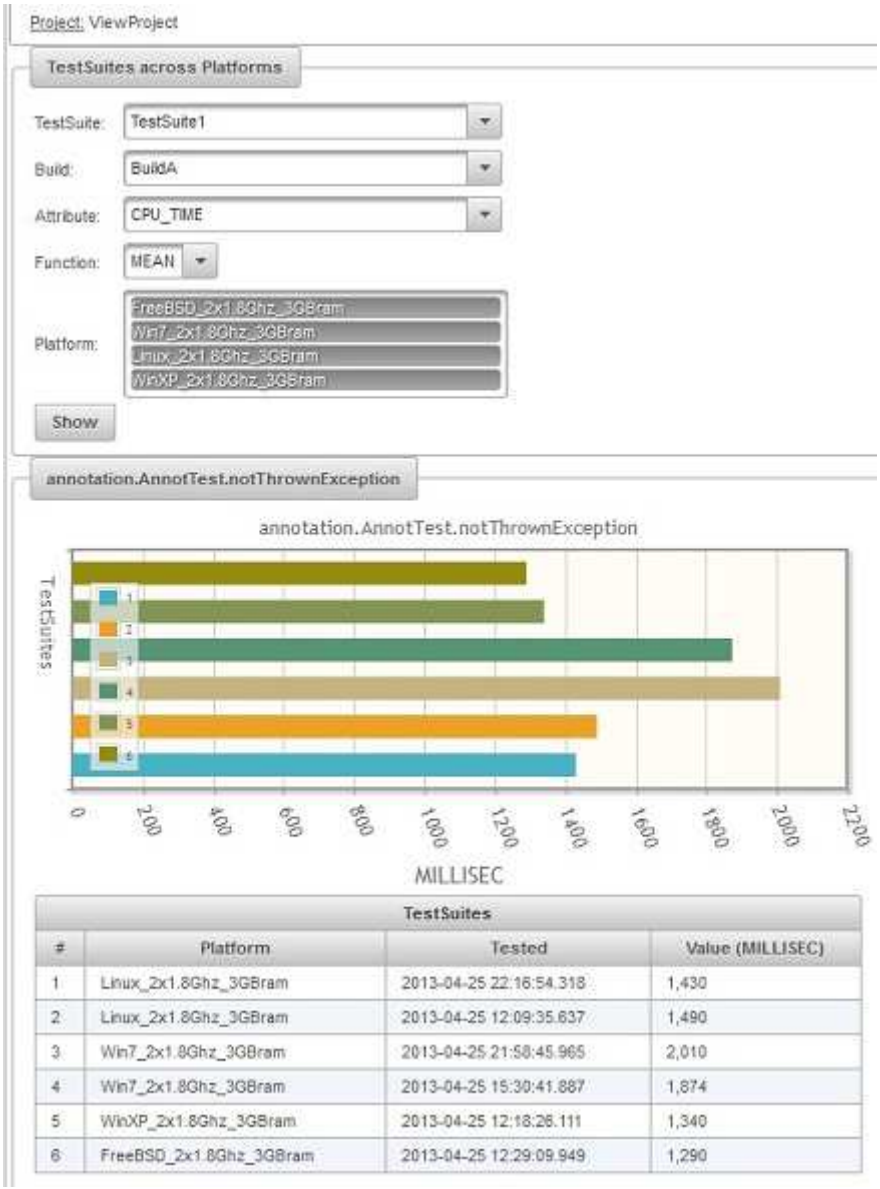
Příloha B – Diagram tříd klientské aplikace



Příloha C – Databázové schéma serverové aplikace

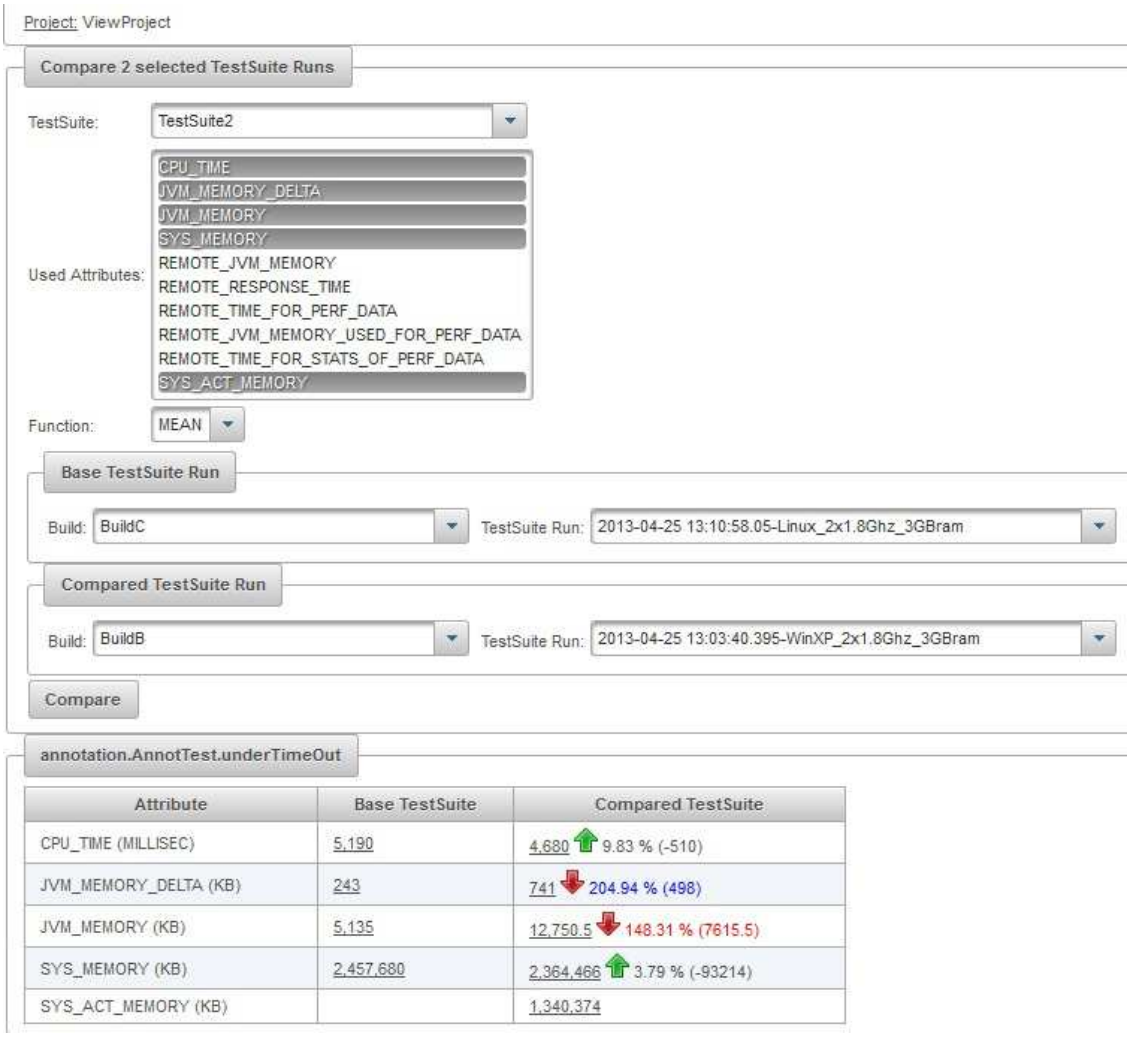


Příloha D – Ukázka náhledu na stránku view1.xhtml (pohled typu I)

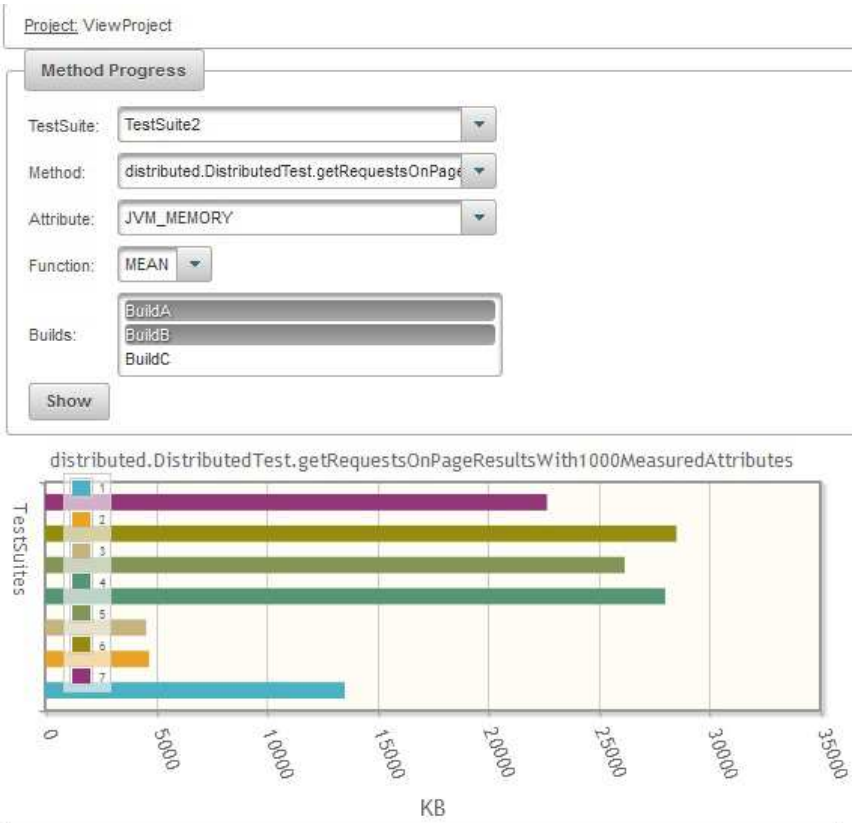


TestSuites			
#	Platform	Tested	Value (MILLISEC)
1	Linux_2x1.8Ghz_3GBram	2013-04-25 22:16:54.318	1,430
2	Linux_2x1.8Ghz_3GBram	2013-04-25 12:09:35.637	1,490
3	Win7_2x1.8Ghz_3GBram	2013-04-25 21:58:45.965	2,010
4	Win7_2x1.8Ghz_3GBram	2013-04-25 15:30:41.887	1,874
5	WinXP_2x1.8Ghz_3GBram	2013-04-25 12:18:26.111	1,340
6	FreeBSD_2x1.8Ghz_3GBram	2013-04-25 12:29:09.949	1,290

Příloha E – Ukázka náhledu na stránku view2.xhtml (pohled typu II)



Příloha F – Ukázka náhledu na stránku view3.xhtml (pohled typu III)



TestSuites				
#	Build	Platform	Tested	Value (KB)
1	BuildA	Linux_2x1.8Ghz_3GBram	2013-04-25 22:16:54.318	13,528.33333
2	BuildA	Win7_2x1.8Ghz_3GBram	2013-04-25 21:58:45.965	4,684.33333
3	BuildA	Win7_2x1.8Ghz_3GBram	2013-04-25 15:30:41.887	4,561.66667
4	BuildA	FreeBSD_2x1.8Ghz_3GBram	2013-04-25 12:29:09.949	27,991.95455
5	BuildA	WinXP_2x1.8Ghz_3GBram	2013-04-25 12:18:26.111	26,166.6
6	BuildB	Linux_2x1.8Ghz_3GBram	2013-04-25 12:17:49.163	28,504.4375
7	BuildA	Linux_2x1.8Ghz_3GBram	2013-04-25 12:09:35.637	22,660.23077

Příloha G – Manuál pro klientskou aplikaci

1. Příprava prostředí pro použití klienta a jeho komponent

Pro použití klienta a jeho komponent je potřeba mít nainstalováno JDK a nástroj Maven. Dále musejí být zkompilovány v následujícím pořadí projekty *PerfObjects*, *PerfClient*, *PerfRtFilter* a *PerfClientServet* (je nezávislý) a umístěny např. do lokálního repozitáře Mavenu, což se provede příkazem *mvn install* v adresáři každého projektu.

2. Klient a jeho standardní listenery

Klient podporuje JUnit testy od verze 4.0 a TestNG testy v jakékoliv verzi. Pro oba tyto frameworky disponuje standardními listenery:

- JUnit: *cz.vutbr.fit.mis.dip.perfclient.listener.junit.PerfListener*
- TestNG: *cz.vutbr.fit.mis.dip.perfclient.listener.testng.PerfListener*

2.1. Implicitně měřené výkonnostní atributy

Každý z listenerů implicitně měří následující výkonnostní atributy.

Atribut	Jednotka	Popis atributu
CPU_TIME	MILLISEC	Procesorový čas spotřebovaný testem.
JVM_MEMORY_DELTA	KB	Změna JVM paměti před a po testu.

2.2. Automatické výkonnostní atributy

Standardní listenery jsou rozšířeny ještě o automatické výkonnostní atributy, které se měří vždy při spuštění testu a po jeho skončení, ale mimo to mohou být měřeny i kdykoliv během testu. Toto rozšíření listenerů je realizováno prostřednictvím třídy *Config* (konfigurační třída listeneru), která definuje automatické výkonnostní atributy uvedené v tabulce níže.

Atribut	Jednotka	Popis atributu
JVM_MEMORY	KB	Aktuální využitá JVM paměť.
SYS_MEMORY	KB	Aktuální využitá systémová paměť.

3. Klient a uživatelem definovaný listener

Klient může používat i uživatelem definovaný listener. Ten musí dědit z jedné ze tříd:

- JUnit: *cz.vutbr.fit.mis.dip.perfclient.listener.junit.BasePerfListener*
- TestNG: *cz.vutbr.fit.mis.dip.perfclient.listener.testng.BasePerfListener*

3.1. Inicializace listeneru konfigurační třídou

Pro inicializaci definovaného listeneru musí být v jeho konstruktoru volána metoda s názvem *initialize*, jíž se parametrem předá instance jeho konfigurační třídy, ve které jsou zpravidla definovány automatické výkonnostní atributy.

Konfigurační třídu listeneru lze vytvořit děděním jedné ze tříd:

- *cz.vutbr.fit.mis.dip.perfclient.config.Config* – Tato třída dědí ze třídy níže a definuje dříve uvedené automatické výkonnostní atributy JVM_MEMORY a SYS_MEMORY.
- *cz.vutbr.fit.mis.dip.perfclient.config.BaseConfig* – V této třídě nejsou definovány žádné automatické výkonnostní atributy.

3.2. Ukázka definice konfigurační třídy a uživatelského listeneru

Vytvoření konfigurační třídy listeneru s názvem *UserConfig* děděním ze třídy *Config*:

```
public class UserConfig extends Config {
	public UserConfig() {
		// system memory except for buffers
		addAutoAttribute(new Att(A.SYS_ACT_MEMORY, U.KB) {
			@Override
			public double gainValue(){
				double value = 0.0;
				try {
					value = sigar.getMem().getActualUsed() / KB_CONST;
				} catch (SigarException e) {
					e.printStackTrace();
				}
				return value;
			}
		});
	}
}
```

Třída *UserConfig* nyní rozšiřuje automatické výkonnostní atributy třídy *Config* o atribut A.SYS_ACT_MEMORY. Pomocí metody *public void addAutoAttribute(Att att)* může uživatel, dle potřeb měření, přidat samozřejmě další automatické atributy. Analogickým způsobem lze konfigurační třídu listeneru vytvořit také děděním z třídy *BaseConfig*.

Vlastní listener se s použitím konfigurační třídy *UserConfig* vytvoří takto:

```
public class UserJUnitPerfListener extends BasePerfListener {
	public UserJUnitPerfListener() {
		initialize(new UserConfig());
	}
}
```

4. Specifikace listeneru klienta v souboru pom.xml

Standardní listener klienta nebo uživatelem definovaný se s využitím Maven Surefire pluginu specifikuje v souboru *pom.xml* testovaného projektu. V tomto souboru se také nastavují systémové proměnné, ve kterých se klientovi předává základní URL adresa webových služeb serverové aplikace (tvořena URL adresou serverové aplikace a řetězcem */rest/rep*), kterým bude klient zasílat výkonnostní data, a dále název testovaného projektu, buildu, test suity, testovací platforma a cesta k adresáři se soubory knihovny SIGAR.

Blíže tato nastavení přibližují soubory *pom.xml* v projektech *JUnitTests* a *TestNGTests* v adresáři test na přiloženém DVD.

5. Maven závilost pro klienta

Aby bylo možno v testovaném Maven projektu použít klienta, je potřeba do souboru *pom.xml* projektu přidat jeho závislost:

```
<dependency>
	<groupId>cz.vutbr.fit.mis.dip</groupId>
	<artifactId>PerfClient</artifactId>
	<version>0.0.1-SNAPSHOT</version>
</dependency>
```

Touto závislost se do projektu vloží jar souboru klienta a klienta již lze v projektu používat.

6. Měření výkonu klientem

Klient implicitně v každém testu projektu měří atributy CPU_TIME, JVM_MEMORY_DELTA a všechny definované automatické výkonnostní atributy.

V případě, že je potřeba provádět výkonnostní měření i uvnitř testů je vhodné do dané třídy vložit atribut pro klienta a inicializovat jej pomocí příkazu *new PerfMonitor()*. Tímto příkazem zároveň určíme, že klient bude pracovat v lokálním módu. Po této inicializaci klienta, již lze plně využívat jeho API pro sběr a odesílání změřených výkonnostních dat.

7. Logování činnosti klienta

Aby byla logována činnost klienta je potřeba do adresáře *src/test/resources* testovaného projektu vložit soubor *log4j.properties* s nastavením pro logovací knihovnu LOG4J.

8. Testování vzdálených enterprise aplikací

Pro výkonnostní testování vzdálené enterprise aplikace je potřeba do jejího *pom.xml* souboru vložit kromě závislosti pro klienta také závislost pro modul *PerfClientServlet*, jehož prostřednictvím bude vzdálený klient inicializován tak, aby mohl změřená výkonnostní data odesílat serverové aplikaci.

8.1. Definování modulu PerfClientServlet

Závislost pro modul *PerfClientServet* je následující:

```
<dependency>
	<groupId>cz.vutbr.fit.mis.dip</groupId>
	<artifactId>PerfClientServlet</artifactId>
	<version>0.0.1-SNAPSHOT</version>
</dependency>
```

PerfClientServlet musí být dále definován v souboru *web.xml* enterprise aplikace a musí být zadána jeho relativní url adresa, jak je ukázáno zde:

```
<servlet>
	<servlet-name>PerfClient Servlet</servlet-name>
	<servlet-class>cz.vutbr.fit.mis.dip.perfservlet.PerfClientInitServlet
	</servlet-class>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>PerfClient Servlet</servlet-name>
	<url-pattern>/init</url-pattern>
</servlet-mapping>
```

8.2. Inicializace vzdáleného klienta

Inicializaci klienta ve vzdálené enterprise aplikaci provádí vždy klient v lokálním módu metodou s názvem *initializeRemotePerfMonitor*. Více je tato metoda popsána v části o API klienta.

8.3. Vzdálený klient a jeho vytvoření

Jen nutno, aby klient ve vzdálené enterprise aplikace pracoval ve vzdáleném módu. Klient v tomto módu se vytvoří příkazem *new PerfMonitor().asRemote() nebo new PerfMonitor().asRemote(new UserConfig())*, kdy mu ve třídě *UserConfig* předáme námi definované automatické výkonnostní atributy. Takto vytvořený klient poté může bezpečně využívat své API.

8.4. Měření response time u vzdálené enterprise aplikace

Pro měření response time u enterprise aplikace je nutno do jejího *pom.xml* souboru vložit již zmíněnou závilost pro modul *PerfClientServlet* a také závislost pro modul *PerfRtFilter*, která vypadá následovně:

```
<dependency>
	<groupId>cz.vutbr.fit.mis.dip</groupId>
	<artifactId>PerfRtFilter</artifactId>
	<version>0.0.1-SNAPSHOT</version>
</dependency>
```

Modul *PerfRtFilter* je dále nutno definovat v souboru *web.xml*, kde se také specifikuje URL vzor pro názvy webových stránek aplikace, u kterých se bude response time měřit. Tato definice modulu *PerfRtFilter* může vypadat například takto:

```
<filter>
	<filter-name>PerfRtFilter</filter-name>
	<filter-class>cz.vutbr.fit.mis.dip.perffilter.RtFilter</filter-class>
</filter>
<filter-mapping>
	<filter-name>PerfRtFilter</filter-name>
	<url-pattern>/faces/results.xhtml</url-pattern>
</filter-mapping>
```

9. API klienta

- **public PerfMonitor()** – Tato metoda vytváří instanci klienta, která bude pracovat v lokálním módu.
- **public PerfMonitor asRemote()** – Metoda uvede klienta do vzdáleného módu, ve které klient musí být, pokud provádí měření na jiném počítači, než jsou spuštěny jednotkové testy. Po zavolání této metody nemá klient definovány žádné automaticky měřené výkonnostní atributy.
- **public PerfMonitor asRemote(BaseConfig config)** – Obdoba předchozí metody, která ve třídě *BaseConfig* či jejím potomku, definuje automaticky měřené výkonnostní atributy.
- **public void initializeRemotePerfMonitor(final String remotePerfMonitorURL)** – Metoda slouží pro inicializaci vzdáleného klienta, kdy mu s použitím modulu *PerfClientServlet* předá ID aktuálně testované test suity a URL adresu serverové aplikace, které má klient zasílat naměřená výkonnostní data. URL adresa testované enterprise aplikace, která se předává této inicializační metodě prostřednictvím parametru, musí být ještě doplněna o relativní URL adresu servletu aplikace *PerfClientServlet*. Inicializaci je vhodné provést v metodě s anotací *@BeforeClass* u JUnit testů nebo *@BeforeSuite* u TestNG testů, čímž se

zajistí, že vzdálený klient bude inicializován ještě předtím, než bude měřit a odesílat výkonnostní data serverové aplikaci.

- **public void gainAttributes()** – Získá hodnoty automatických atributů a uloží je do bufferu klienta.
- **public void gainAttributes(MAtt[] attrs)** – Získá hodnoty automatických atributů a spolu s manuálními atributy předanými v parametru *attrs* je uloží do bufferu klienta.
- **public void setAttributes(MAtt[] attrs)** – Uloží manuální atributy předané v parametru *attrs* do bufferu klienta.
- **public void sendAttributes()** – Odešle obsah bufferu klienta s výkonnostními daty serverové aplikaci.
- **public void sendAttributes(MAtt[] attrs)** – Metoda funguje stejně jako metoda s názvem *sendAttributes*, jen jsou s výkonnostními daty odeslány také manuální atributy předané v parametru *attrs*.
- **public void gainAttributesAndSend()** – Získá hodnoty automatických atributů a uloží je do bufferu klienta. Poté je celý obsah bufferu odeslán serverové aplikaci.
- **public void gainAttributesAndSend(MAtt[] attrs)** – Metoda funguje stejně jako metoda s názvem *gainAttributesAndSend*, jen jsou s výkonnostními daty odeslány také manuální atributy předané v parametru *attrs*.

Příloha H – Manuál pro serverovou aplikaci

1. Nasazení serverové aplikace PerfServer

1.1. Co mít nainstalováno

Je nezbytné mít nainstalováno:

- JDK
- Maven
- PostgreSQL
- JBoss AS 7

1.2. Příprava spojení s databází

V prostředí PostgreSQL vytvoříme databázi, jež bude naše aplikace *PerfServer* používat. Do adresáře JBoss AS poté zkopírujeme JDBC ovladač pro PostgreSQL. S použitím tohoto ovladače a vytvořené databáze následně na serveru definujeme tzv. data source, který musí být rovněž definován v soboru *persistence.xml* aplikace *PerfServer*.

1.3. Překlad a nasazení aplikace PerfServer

Před nasazením aplikace *PerfServer* na JBoss AS, je nezbytné přeložit projekt *PerfObjects* příkazem *mvn install*. Poté již lze přeložit a nasadit aplikaci *PerfServer* na JBoss AS příkazem *mvn jboss-as:deploy*. Před touto operací musí být JBoss AS samozřejmě spuštěn.

2. Konfigurace aplikace PerfServer před zahájením testování

Aplikace *PerfServer* u jakýchkoliv příchozích výkonnostních dat od klienta ověřuje, že všechny výkonnostní atributy včetně jejich jednotek použité v těchto datech jsou na její straně identicky definovány. Před zahájením testování je tedy nutné očekávané výkonnostní atributy spolu s jejich jednotkami v aplikaci *PerfServer* definovat. Jednotky se definují na stránce *units.xhtml* a atributy na stránce *attributes.xhtml*.

V případě, že bude konfigurace výkonnostních atributů aplikace *PerfServer* v nesouladu s příchozími daty, všechna tato data budou aplikací *PerfServer* ignorována a klientovi bude zaslána zpráva s HTTP kód s číslem 333 spolu s popisem konfiguračního problému.

Ukázka interpretace takové zprávy klientem je na následující straně:

```
2013-04-25 14:35:06 DEBUG PerfMonitor:117 - SEND PERFORMANCE DATA
2013-04-25 14:35:06 WARN  Communicator:50 - CODE: 333
2013-04-25 14:35:06 WARN  PerfMonitor:32 - Attribute REMOTE_RESPONSE_TIME is not
defined on the server.
2013-04-25 14:35:06 WARN  PerfMonitor:34 - Unit MB used for attribute JVM_MEMORY is
not defined on the server.
2013-04-25 14:35:06 WARN  PerfMonitor:38 - Attribute SYS_ACT_MEMORY and its unit MB
are not defined on the server.
```

3. Návratové kódy webových služeb aplikace PerfServer

Návratové kódy odpovídají svým významem standardním návratovým kódům v protokolu HTTP. Některé kódy mají však v aplikaci speciální význam:

HTTP kód	Význam
333	Výkonnostní atribut či jeho jednotka není v aplikaci definován.
399	Při přijetí výkonnostních dat od klienta nebyla testována žádna metoda. Tato data jsou zahozena, protože je nelze asociovat s žádnou metodou.

Význam kódu ve zprávách je vždy navíc popsán polem *Message* v hlavičce těchto zpráv. Hodnota tohoto pole blíže specifikuje vykonanou činnost webovou službou či vzniklý problém, který mohl při této činnost nastat. Další vlastní HTTP kódy jsou uvedeny v části o exportu data pro portál qVue.

4. Export dat pro portál qVue

Export dat odpovídá svým charakterem srovnávacímu pohledu typu II na výkonnostní data.

4.1. Získání dat pro export

Data pro export lze získat vysláním HTTP požadavku typu GET na URL adresu v následujícím formátu:

SERVER_URL/rest/rep/export/PROJECT;testsuite=TEST_SUITE;basebuild=BASE_BUILD;build=BUILD

- **SERVER_URL** – Značí URL adresu aplikace *PerfServer*.
- **PROJECT** – Název projektu, v rámci nějž budou použity zadané buildy níže.
- **TEST_SUITE** – Název test suity, která bude použita pro porovnávání.
- **BASE_BUILD** – Jméno buildu, ze kterého bude vzata referenční test suita s názvem TEST_SUITE. Vždy se použije test suita, která byla naposledy dotestována bez ohledu na platformu, na které testování probíhalo.

- **BUILD** – Má stejný význam jako BASE_BUILD jen bude test suita brána jako srovnávaná (bude tedy srovnávána vůči test suitě z buildu s názvem BASE_BUILD).

V případě, že BASE_BUILD a BUILD jsou totožné buildy, jako referenční test suita bude použita ta, která byla v daném buildu dotestována jako předposlední. Jako test suita, která bude vůči ní srovnávána, bude vzata ta, které byla dotestována jako poslední.

4.2. Vlastní HTTP chybové kódy pro export

V tabulce níže jsou uvedeny vlastní HTTP kódy, které mohou být při exportu dat vráceny, v případě vzniku nějakého problému.

HTTP kód	Význam
340	Projekt s názvem PROJECT neexistuje.
341	Build projektu s názvem BUILD neexistuje.
342	Build projektu s názvem BASE_BUILD neexistuje.
343	Srovnávaná test suita s názvem TEST_SUITE buildu s názvem BUILD neexistuje.
344	Referenční test suita s názvem TEST_SUITE buildu s názvem BASE_BUILD neexistuje.

5. Webové stránky aplikace PerfServer

Zde jsou uvedeny všechny webové stránky aplikace *PerfServer* včetně stručného popisu jejich funkcionality:

- **index.xhtml** – Stránka zobrazuje aktuálně testované test suity v rámci všech projektů.
- **recently_testsuites.xhtml** – Zobrazuje všechny test suity, jejichž testování bylo dokončeno v uživatelem stanoveném časovém období. Implicitně jsou zobrazeny test suity dotestované v posledních deseti dnech.
- **projects.xhtml** – Stránka zobrazuje všechny testované projekty. Projekty lze zde mazat. Při smazání projektu budou smazány všechny jeho data.
- **uploadfile.xhtml** – Import výsledků výkonnostního testování test suity prostřednictvím XML souboru. Formát tohoto souboru je definován XML schématem v souboru *src/main/webapp/resources/files/schema.xml* aplikace.
- **thresholds.xhtml** – Správa globálních hraničních omezení pro výkonnostní atributy.
- **attributes.xhtml** – Správa podporovaných výkonnostní atributů.
- **units.xhtml** – Správa podporovaných jednotek atributů.

- **local_thresholds.xhtml** – Správa hraničních omezení pro výkonnostní atributy u zvoleného projektu.
- **testsuites.xhtml** – Stránka zobrazuje všechny test suity v rámci zvoleného projektu. Test suity lze zde mazat. Při smazání test suity budou smazány všechny její metody včetně výkonnostních dat.
- **methods.xhtml** – Zobrazuje všechny metody u zvolené test suity a to včetně těch, které byly uživatelem při testování pomocí anotací ignorovány. U metod jsou zobrazeny také hodnoty parametrů, se kterými byly volány (podporováno jen u TestNG), a výsledky metod. U metod s chybou lze po najetí myší na ikonu výsledku zobrazit také příslušnou výjimku.
- **results.xhtml** – Na stránce je zobrazena informační a výkonnostní sekce u zvolené metody. Informační část může obsahovat popis metody (podporováno jen u TestNG) a případnou výjimku, se kterou metoda skončila. Výkonnostní část obsahuje záložky s jednotlivými měřenými výkonnostními atributy. Na záložce je zobrazen graf vývoje hodnot atributu v závislosti na čase, tabulka s těmito hodnotami a souhrnné statistické údaje určené z těchto dat.
- **view1.xhtml** – Realizuje srovnávací pohled na výkonnostní data typu I. Grafy v pohledu umožňují po kliknutí na konkrétní hodnotu výkonnostního atributu zobrazit jeho detailní průběh na stránce *results.xhtml*.
- **view2.xhtml** – Realizuje srovnávací pohled na výkonnostní data typu II.
- **view3.xhtml** – Realizuje srovnávací pohled na výkonnostní data typu III. Graf v pohledu je interaktivní stejným způsobem jako grafy na stránce *view1.xhtml*.

Příloha I – Obsah přiloženého DVD

Přiložené DVD obsahuje následující adresářovou strukturu:

- **text**
  - *dp_vlasak.doc* – text diplomové práce ve formátu doc
- **app**
  - *PerfObjects* – Projekt, který obsahuje implementace tříd použitých pro přenos dat mezi klientskou a serverovou aplikací. Kromě těchto tříd jsou zde také definovány konstanty sdílené klientskou i serverovou aplikací (relativní URL adresy webových služeb apod.).
  - *PerfClient* – klientská aplikace
  - *PerfClientServlet* – inicializační servlet pro vzdáleného klienta
  - *PerfRtFilter* - Nástroj pro měření a odesílání response time serverové aplikaci.
  - *PerfServer* – serverová aplikace
  - *TestedPerfServer* – modifikovaná serverová aplikace pro účely testování
- **tests**
  - *DataForComplexTest* – Projekt s TestNG testy, jejichž účelem je ověřit funkčnost API klienta pro získávání automatických a manuálních výkonnostních atributů, a schopnost klienta tyto data odeslat serverové aplikaci. Data odeslaná při těchto testech byla použita v distribuovaných testech projektů JUnitTests a TestNGTests.
  - *JUnitTests* – Projekt s JUnit testy, které kompletně ověřují funkčnost klienta v JUnit testech. Součástí testů jsou i komplexní distribuované testy.
  - *TestNGTests* – Projekt s TestNG testy, které jsou ekvivalentní testům v projektu JUnitTests.
- **examples**
  - *ClientLog.txt* – Ukázka logování komunikace klienta se serverovou aplikací.
  - *qVue.txt* – Ukázka exportovaných výkonnostních dat pro portál qVue.
  - *TestSuite.xml* – Příklad XML souboru pro import výkonnostních dat test suity.
- **sigarLib** – Adresář, který obsahuje soubory knihovny SIGAR.